Ruby on Rails

Coloque sua aplicação web nos trilhos

Casa do Código

VINÍCIUS B. FUENTES

Agradecimentos

Monsters are real, and ghosts are real too. They live inside us, and sometimes, they win.
– Stephen King

Este livro não existiria sem a ajuda dos meus grandes amigos Matheus Bodo, Willian Molinari, Sérgio Schezar e Vinícius Uzêda, que me acompanharam nesse processo quase todos os dias, revisando, criticando e opinando minuciosamente o conteúdo desse livro. Muito obrigado!

Muito obrigado também a família Casa do Código e Caelum, pela oportunidade de escrever esse livro e pelos ensinamentos, especialmente ao Adriano Almeida, pelo difícil trabalho de colocar ordem às minhas palavras.

Agradecimentos especiais também ao GURU-SP (Grupo de Usuários Ruby de São Paulo), à PlataformaTec e aos amigos do ICMC-USP, pois se sei alguma coisa, devo tudo a eles.

Agradeço também a minha família e amigos, pela força e por tolerarem meses sem notícias enquanto me mudo para outro país.

Por fim, agradeço principalmente a você leitor, por investir seu tempo a aprender uma tecnologia que eu pessoalmente gosto tanto. Espero sinceramente que seja uma jornada divertida e lucrativa ao mesmo tempo!

Sumário

Capítulo 1

Introdução

Quem, de três milênios, / Não é capaz de se dar conta / Vive na ignorância, na sombra, / À mercê dos dias, do tempo.
– Goethe

Se você desenvolve para a Web, provavelmente já ouviu falar sobre Ruby on Rails. O Ruby on Rails (ou também apenas "Rails") é um *framework open-source* para desenvolvimento de aplicações web, criado por David Heinemeier Hansson (ou "DHH"). O *framework*, escrito na linguagem Ruby, foi extraído de um produto de sua empresa, o Basecamp® (http://basecamp.com) em 2003.

Desde então ele ficou muito famoso, levando também a linguagem Ruby, anteriormente apenas conhecida no Japão e em poucos lugares dos Estados Unidos, ao mundo todo. Mas por que cresceu tanto? O que a linguagem e o *framework* trouxeram de novo para sair do anonimato e praticamente dominar o mercado de *startups* nos Estados Unidos e no mundo?

Vamos primeiro à linguagem Ruby. A linguagem Ruby foi criada pelo extremamente talentoso programador Yukihiro "Matz" Matsumoto. O objetivo dele ao criar

o Ruby em 1995 foi a felicidade e o prazer de quem programa, ao invés de performance ou algum outro aspecto técnico. Ele usou Perl como inspiração, com várias pitadas de Smalltalk. Veremos exemplos dessas influências, e também o que significa esta "felicidade para o programador" no capítulo 2.

Quanto ao Rails, na época em que foi lançado, trouxe uma visão diferente ao desenvolvimento Web. Naquele momento, desenvolver para Web era cansativo, os *frameworks* eram complicados ou resultavam em sistemas difíceis de se manter e de baixa qualidade.

O DHH, ao desenvolver o Basecamp, pensou principalmente nos seguintes aspectos:

- "Convention over configuration", ou convenção à configuração: ao invés de configurar um conjunto de arquivos XML, por exemplo, adota-se a convenção e apenas muda-se o que for necessário;
- "Dont Repeat Yourself", ou "não se repita": nunca você deve fazer mais de uma vez o que for necessário (como checar uma regra de negócio);
- Automação de tarefas repetidas: nenhum programador deve perder tempo em tarefas repetitivas e sim investir seu tempo em resolver problemas interessantes.

Esses conceitos são amplamente explorados no clássico *The Pragmatic Programmer: From Journeyman to Master [1]*, leitura recomendada. Esta soma de tecnologias e práticas são bastante prazerosas de se trabalhar; é possível realizar muito com pouco tempo e você verá, capítulo a capítulo, como essas ideias são representadas em todos os aspectos do *framework*.

1.1 Para quem é este livro

O objetivo deste livro é apresentar um pouco da linguagem Ruby e aprender os primeiros passos a desenvolver com o *framework* Ruby on Rails. Mais além, vamos aprender a desenvolver aplicativos **com** Rails. Algumas vezes usaremos inclusive componentes feitos em Ruby puro. Esses momentos são extremamente importantes para aprendermos também algumas boas práticas.

Tendo isso em mente, este livro serve para pessoas que:

- Não conhecem ambas as tecnologias, ou conhecem apenas Ruby, mas desejam conhecer o Rails;

- Já conhecem Rails, mas não estão confortáveis em como fazer aplicações bem organizadas;
- Já conhecem Rails superficialmente, mas querem aprimorar conhecimentos e boas práticas.

1.2 Organização

Este livro foi construído em três partes. A primeira parte é dedicada a entender a linguagem Ruby. Nessa parte vamos aprender desde a sintaxe até boas práticas com a linguagem usando exemplos práticos.

A segunda parte é dedicada a entender o contexto que o Ruby on Rails trabalha. Essa é a parte teórica, na qual vamos entender quais são os principais conceitos por trás do *framework*. Vamos ver também, em alto nível, quais são os componentes do Rails e como eles se relacionam.

A terceira parte é onde vamos fazer a aplicação com Rails. Passo a passo, vamos implementar uma aplicação do início ao fim e, durante a construção de cada funcionalidade, aprender como juntar as partes do *framework*. Vamos revisitar funcionalidades, aprimorando-as, e mostrar como é o processo de criação de uma aplicação real, de forma que você aprenda uma das possíveis maneiras de construir suas aplicações no futuro.

Vamos construir um aplicativo chamado Colcho.net. O Colcho.net é um site para você publicar um espaço sobrando na sua casa para hospedar alguém por uma ou mais noites. O site vai ter:

- Cadastro de usuário, com encriptação de senha;
- Login de usuários;
- Envio de emails;
- Internacionalização;
- Publicação e administração de quartos;
- Avaliação de quartos e *ranking*;
- Busca textual;
- *URL slugs*;

- Uploads e thumbnails de fotos.

Embora este livro tenha sido construído de forma que a leitura progressiva seja fácil, ele pode servir como consulta. Vamos dividir o sistema que será construído em algumas funcionalidades principais e atacá-las individualmente em cada capítulo.

Parte I

A linguagem Ruby

Capítulo 2

Conhecendo Ruby

Não reze por uma vida fácil, mas sim para ter forças para uma vida difícil.
– Bruce Lee

Ruby é uma linguagem de script muito interessante. Como vimos na Introdução, Ruby tem como parentesco Perl e várias pitadas de Smalltalk, além de outras características "Lispianas".

Isso se reflete de várias formas na linguagem. Primeiramente, Ruby é uma linguagem dinamicamente "tipada", ou seja, não precisamos declarar os tipos dos objetos e variáveis, característica comum de algumas linguagens de script, como Python e PHP. Ruby também é uma linguagem orientada a objetos, porém com propriedades não muito comuns. Uma delas, por exemplo é o fato de que, como em Smalltalk, chamadas de métodos nada mais são do que envio de mensagens a objetos, e isso é refletido na forma que podemos implementar programas. Para demonstrar esses recursos, vamos a um exemplo onde chamamos um método em um objeto:

```
shopping_cart.clear
```

O mesmo método pode ser chamado da seguinte forma:

```
shopping_cart.send 'clear'
```

Neste trecho de código, estamos enviando uma mensagem de nome 'clear', que por sua vez chama o método `clear` do objeto `shopping_cart`. Note também o uso de `snake_case`, ao invés de `camelCase`.

O que é camelCase e snake_case?

CamelCase e snake_case são duas formas bastante populares de escrever código. Na forma CamelCase, diferenciamos cada palavra via letras maiúsculas, sem separar as palavras. Já o snake_case, deixamos todas as palavras em minúsculas as separamos usando underscore (_).

A plataforma Ruby também oferece ao programador várias ferramentas para que um programa possa descobrir informações sobre si mesmo. Por exemplo, é possível descobrir se uma constante foi declarada ou se um objeto responde a um método. Veja o seguinte exemplo, no qual verificamos se o objeto `shopping_cart` responde ao método `clear`:

```
shopping_cart.respond_to? 'clear' # => true
```

Query methods

Query methods são métodos terminados em ?. Eles essencialmente só devem ser usados quando queremos saber se o resultado é verdadeiro ou falso, independentemente do seu resultado de fato. São usados basicamente com `if`, `unless` e afins.

Neste exemplo, o próprio programa verifica se o objeto responde ao método `clear`. Isso pode parecer estranho no início para quem nunca viu uma linguagem reflexiva antes, mas isso dá bastante poder ao programador que deseja fazer soluções rebuscadas.

Neste capítulo, iremos aprender um pouco da linguagem Ruby para que seja possível entender e criar aplicações simples em Rails. É importante ressaltar que é ainda necessário aprender mais da linguagem, pois este livro vai apenas te ensinar o básico

o suficiente para dar o pontapé inicial. Mas para sermos bons programadores, precisamos sempre saber a fundo a linguagem que trabalhamos. No capítulo 14 "Próximos Passos" você saberá onde buscar mais informações depois de terminar este livro.

2.1 Instalação do Ruby e Rails

Existem diversas implementações de Ruby, como JRuby (http://jruby.org/) e Rubinius (http://rubini.us), mas iremos usar a versão MRI, ou Matz Ruby Interpreter, a implementação canônica de Ruby, criada pelo autor original.

Instalação no OSX

Para quem está iniciando com Ruby on Rails, a forma mais simples é usar o RVM, ou Ruby enVironment Manager. Antes de instalar o RVM, porém, é necessário instalar os pacotes de desenvolvimento da Apple.

Se você não tem o Xcode, vá na página do Apple Developer Tools (https://developer.apple.com/downloads/index.action) e baixe o 'Command Line Tools for Xcode', versão mais atual e siga os passos de instalação. Se tiver o Xcode mais recente, você pode abri-lo, ir nas Preferências > Downloads e instalar o 'Command Line Tools'.

Depois de instalar o Command Line Tools, basta executar o seguinte comando no terminal:

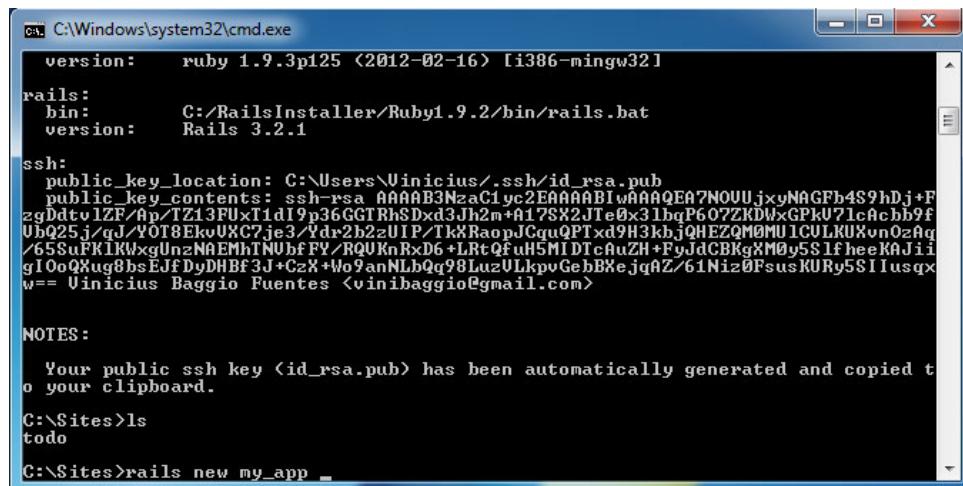
```
curl -L get.rvm.io | bash -s stable --rails
source $HOME/.rvm/scripts/rvm
```

Aproveite e vá tomar um café, vai demorar um pouco. Ao finalizar, este comando deixará instalado tudo que você precisa para começar a desenvolver com Ruby on Rails.

Instalação no Linux

Tanto quanto usuários OSX, a forma mais simples de começar com Ruby on Rails no Linux é usar o RVM, ou Ruby enVironment Manager.

Primeiramente é necessário instalar os pacotes de desenvolvimento do seu sistema. Procure o manual da sua distribuição para saber como instalar. No Ubuntu, por exemplo, basta instalar o pacote `build-essential` e mais algumas outras bibliotecas de dependências:

```
sudo apt-get install build-essential libreadline-dev libssl-dev curl \
libsqlite3-dev
```

Em seguida, basta executar:

```
curl -L get.rvm.io | bash -s stable --rails
source $HOME/.rvm/scripts/rvm
```

Este comando irá instalar o RVM e também, automaticamente, irá instalar a versão mais atual do Ruby e do Rails e todas as dependências. Aproveite para tomar um café ou um chá, pois demora um pouco.

Instalação no Windows

Para instalar o ambiente de Ruby e Rails no Windows, o jeito mais fácil é usar o RailsInstaller (http://www.railsinstaller.org), que já instala o Ruby versão MRI e todas as dependências do Rails. Para instalar o ambiente, basta baixar o instalador e seguir as instruções apresentadas. Ao completar a instalação, você terá um console para executar comandos, como observado na imagem 2.1.

Figura 2.1: Console do RailsInstaller

Para acessá-lo novamente, basta usar o atalho 'RailsInstaller > Command Prompt with Ruby on Rails'.

2.2 Primeiros passos com Ruby

Para começar, o IRB, ou *Interactive Ruby Shell* é onde é possível executar pequenos trechos de código. Recomenda-se o uso do IRB para testar pequenas ideias ou sintaxe da linguagem, para que, no futuro, seja incorporado dentro de um programa completo.

Ao abri-lo, você vai se deparar com um terminal parecido com o seguinte:

```
irb(main):001:0>
```

Nele, você pode digitar trechos de código ruby. Ele vai ser avaliado e o retorno dessa linha é impresso em seguida:

```
irb(main):001:0> a = "0"
=> "0"
irb(main):002:0> a
=> "0"
```

Para sair, basta digitar `exit`.

> **Notação**
>
> Para facilitar a leitura, a partir de agora não será mais incluso o prompt do IRB nos exemplos de código, porém o resultado da operação será colocado adiante de cada trecho de código, usando um comentário da seguinte maneira:
>
> # => Saída

2.3 Tipos e estrutura de dados

A primeira característica que vemos nessas linhas em Ruby é que a linguagem é de tipagem dinâmica, ou seja, não precisamos declarar nenhum tipo para criar uma variável. Dessa forma, é possível fazer o seguinte:

```
a = "string"
# => "string"

a = 100
# => 100
```

Strings

Strings, tal como outras linguagens de script, são fáceis de serem criadas, com uma notação literal usando aspas duplas ou simples. Porém, como tudo em Ruby é um objeto, é possível chamar métodos diretamente à notação literal:

```
"this is sparta".upcase
# => "THIS IS SPARTA"
```

É também possível compor strings maiores fazendo algumas operações:

```
"hello" + " " + "world"
# => "hello world"
```

O interessante também é que Strings reagem à multiplicação. A multiplicação espera um inteiro como segundo parâmetro:

```
"." * 10
# => ".........."
```

É possível também acumular strings usando <<:

```
greetings = "Hello"
greetings << " "
greetings << "World"

puts greetings # Hello World
```

Interpolação de strings

Outro fato importante de strings é a interpolação, ou a combinação de código para a composição de strings. É importante ressaltar que este tipo de interpolação de strings só funciona com aspas duplas:

```
name = "Pedro"
"Tudo certo, #{name}?" # Tudo certo, Pedro?

'Tudo certo, #{name}?' # "Tudo certo, \#{name}?"
```

A interpolação vai ser avaliada apenas uma vez antes de qualquer outra operação com a string resultante:

```
i = 0
"#{i = i + 1} " * 3 # "1 1 1 " e não "1 2 3 "
```

Há outra maneira mais complexa de fazer interpolação de Strings. Leitores que conhecem C e Java vão se lembrar da função `sprintf` ao fazer interpolação com o operador `%`. Neste caso, é possível usar aspas simples:

```
'Custo total: $%.2f' % 100 # Custo total: $100.00
```

Para a relação completa de como a formatação deste operador funciona, veja a documentação online: http://www.ruby-doc.org/core-1.9.3/Kernel.html#method-i-sprintf.

Acessando caracteres e substrings

Strings, no final das contas, são cadeias de caracteres, como se fossem Arrays. Portanto, é possível acessar caracteres via método `[]` com o índice:

```
a = "hello world"
a[0]  # 'h'
a[6]  # 'w'
a[-1] # "d" - Valores negativos contam do fim ao começo
a[-2] # "l"
```

Este acessor leva em conta o *encoding* atual, portanto, em *encodings* multibyte, ou seja, *encodings* que podem usar mais de um byte para representar um caractere, o acessor irá retornar o caractere em si e não o byte naquela posição:

```
# encoding: utf-8
a = "Olá, tudo bom com você?"
puts a[2]          # á
puts a.bytesize    # 25
puts a.length      # 23
```

Para acessar substrings, usamos um outro tipo de variável, o Range, ou 2 índices indicando o começo e o fim:

```
a = "Hello world"
a[6, 10]  # => "world"
a[6..10]  # => "world"
a[-5..-1] # => "world"
a[-1..-5] # => "" - quando não é possível, o método retorna string vazia.
```

Inteiros e Floats

Inteiros e floats possuem representações literais como em qualquer linguagem:

```
a = 100
a = 100000000    # Fica difícil ler com muitos zeros
a = 100_000_000  # Pode usar _ para separar e possui o mesmo efeito
a = 100.0

100.0.to_i       # 100 - Conversão de float para inteiro
100.to_f         # 100.0 - Conversão de inteiro para float
```

Quando se faz operações entre os dois tipos, especialmente divisões, acontece a coerção de tipos, ou seja, um inteiro é automaticamente convertido para float quando há um float envolvido na conta.

Quando dividimos um inteiro com outro, o resultado será um inteiro truncado, removendo-se a "parte quebrada", ou seja, remove-se os decimais, e converte-se a um inteiro. Por exemplo:

```
100 / 3 # => 33
```

Para obter um valor real/float, é necessário convertê-lo antes para float:

```
100.to_f / 3 # => 33.3333333...336
```

Cuidado com floats!

A representação de números reais usada é o IEEE-754, a forma padrão de representar números reais em binários, usada não só por Ruby, mas também por C, Java e JavaScript. Porém, este padrão possui um problema perigoso de arredondamento:

```
0.0004 - 0.0003 == 0.0001 #=> false
```

Este é um problema sério, pois muitas pessoas não conhecem ou não sabem deste problema e usam floats para valores financeiros, o que é errado. Para isso, usa-se a classe `BigDecimal`, do conjunto de bibliotecas padrão do Ruby. Apesar de ser muito mais lenta que floats, eles são precisos e se tratando de cálculos financeiros, performance geralmente não é um problema, se comparado com aplicações científicas.

Constantes

Constantes em Ruby são todas as variáveis que começam com uma letra maiúscula, independente do restante. Veja o seguinte exemplo:

```
Pi = 3.14159
```

Porém, a comunidade Ruby adota um padrão de nomenclatura. Como classes e módulos também usam constantes para identificá-los, adotamos CamelCase para classes e MAIÚSCULAS para valores, de forma a distingui-los facilmente.

```
EULER = 2.718 # Constante da forma correta
Pi            # Classe?
```

Cuidado! Constantes não são constantes!

Infelizmente é possível alterar valores de constantes no Ruby. Ao tentarmos fazer isso, porém, o interpretador irá exibir uma mensagem de alerta:

```
PHI = 1.618
PHI = 999
# (irb):9: warning: already initialized constant Pi
# => 999

PHI
# => 999
```

Arrays

Arrays em Ruby possuem representação literal, parecido com Python e Perl e também podem conter qualquer tipo de objetos:

```
a = [1, 2, 1+2]     # [1, 2, 3]
a << 4              # [1, 2, 3, 4]
a << "string!"    # [1, 2, 3, 4, "string"]
```

O comportamento de Arrays é bem parecido com strings, ou seja, é possível usar Ranges e o método `[]` para acessar elementos diretamente:

```
a = ["a", "b", "c", "d", "e"]
a[0]                                # 'a'
a[0..2]                             # ["a", "b", "c"]
a[0, 2]                             # ["a", "b"] - Atenção a este exemplo!
```

É possível fazer atribuição de elementos de Array de uma forma bastante conveniente:

```
list = ["a", "b", "c"]
first, second = list

first  # => "a"
second # => "b"
```

Usando o *splat operator*, é possível obter o restante de toda a lista:

```
list = ["a", "b", "c"]
first, *tail = list

first # => "a"
tail  # => ["b", "c"]
```

Array de strings são chatas de digitar, portanto o Ruby tem uma maneira para facilitar isso:

```
a = %w{a b c d e}
a #  ["a", "b", "c", "d", "e"]

# Porém, palavras com espaços são problemáticas
b = %w{"long words" small tiny}
# ["\"long", "words\"", "small", "tiny"]

# Palavras com espaços devem ser "escapadas"
c = %w{long\ words small tiny}
# ["long words", "small", "tiny"]
```

Arrays também fazem operações matemáticas:

```
a = ["a", "b", "c"]
b = ["a", 2, 3]
```

```
a + b # ["a", "b", "c", "a", 2, 3]
a - b # ["b", "c"]
b - a # [2, 3]

c = [1, 2]
c * 3 # [1, 2, 1, 2, 1, 2]
```

Além dessas operações mais canônicas, Arrays ainda suportam interseção e união de conjuntos:

```
a = [1, 2, 3]
b = [3, 4, 5]

a + b    # [1, 2, 3, 3, 4, 5]
a | b    # [1, 2, 3, 4, 5] - Na união, elementos duplicados são removidos
a & b    # [3] - Na interseção, apenas os repetidos ficam
```

Arrays não deixam de ser objetos em Ruby, portanto possuem vários métodos interessantes. No exemplo abaixo vemos o uso dos métodos mais comuns:

```
[1, 2, 3].reverse
# [3, 2, 1] - inverte o array

['acerola', 'laranja'].join(' e ')
# "acerola e laranja" - concatena strings com o parâmetro passado

[10, 20, nil, '', false, true].compact
# [10, 20, '', false, true] - remove nils

[6, 3, 9].sort # [3, 6, 9] - ordena os resultados

[3, 3, 9].uniq # [3, 9] - apenas os elementos únicos

[[3], 2, 1].flatten # [3, 2, 1] - achata listas internas

a = [1, 2, 3]
a.pop   # 3 - sai o último elemento
a       # [1,2]

a.shift # 1 - sai o primeiro elemento
a       # [2]
```

Note que a maioria desses métodos possui a versão *bang*, ou seja, terminada em exclamação, que modifica o conteúdo da própria variável, ao invés de retornar uma cópia modificada:

```
a = [[3], 2, 1]
b = a.flatten

a
# => [[3], 2, 1]

b
# => [3, 2, 1]

b = a.flatten!

a
# => [3, 2, 1],
b
# => [3, 2, 1]
```

Métodos bang (!)

Em Ruby, é possível criar métodos que terminam em ! e são usados em principalmente dois casos:

1) Métodos que modificam estado interno do objeto, como vimos no exemplo anterior;

2) Métodos que, ao falharem, disparam uma exceção.

No uso de métodos do Rails é possível observar o caso em que a versão "bang" do método resulta em uma exceção:

```
Post.find_by_title 'Unexistent post' # => nil

Post.find_by_title! 'Unexistent post'
# ActiveError::RecordNotFound
```

Hashes

Ruby também possui representação literal para Hashes, tal como Python (apesar de serem conhecidos naquelas terras por *dicts* ou dicionários). Hashes são estruturas de dados similares a Array, porém ao invés de acessarmos valores com índices numéricos, podemos usar qualquer objeto:

```
frequency = Hash.new
frequency["hello"] = 1
frequency["world"] = 2
frequency[1] = 10
```

É possível criar hashes também com uma sintaxe mais cômoda:

```
# equivalente ao exemplo anterior
frequency = { "hello" => 1, "world" => 2, 1 => 10 }
```

Ainda é possível usar uma terceira sintaxe, usada para criar hashes contendo símbolos como chaves (veremos mais detalhes sobre símbolos ainda nesta seção). Veja o exemplo a seguir:

```
# Sintaxe clássica para hashes tendo símbolos como chaves
frequency = { :hello => 1,  :world => 2 }

# Nova sintaxe
frequency = { hello: 1, world: 2 }
```

Uso de sintaxe de Hash

Não há nenhuma diferença entre as duas sintaxes de hashes com símbolos, ou seja, elas são puramente estéticas. Neste livro, vamos nos ater à sintaxe clássica apenas para manter consistência com outras hashes, ficando à critério do leitor escolher qual prefere usar. Lembre-se apenas de ser consistente.

Hashes possuem métodos bem úteis. Alguns dos mais usados estão exemplificados em seguida:

```
frequency = { "hello" => 1, "world" => 2 }

frequency.keys
# ["hello", "world"]

frequency.values
# [1, 2]

frenquency.has_key?("hello")
# true

frequency.has_value?(3)
# false
```

Menção honrosa

É comum lidarmos com integrações com outros sistemas e APIs que retornam construções complexas em hashes e quase sempre não são consistentes. Vejamos um exemplo fictício abaixo:

```
user_data = {
  'email' => 'cicrano@example.com',
  'full_name' => 'Cicrano'
}
```

Imaginemos agora que queiramos acessar os atributos `email`, `full_name` e `address`. `address`, porém, não está presente no exemplo de hash acima. Dessa forma, ao fazermos o código abaixo, temos um problema:

```
address = user_data['address']
address.strip
# NoMethodError: undefined method `strip' for nil:NilClass
```

Por esse motivo, é bastante comum fazermos proteção contra `nil` usando uma expressão idiomática. Dessa forma, nosso código fica mais limpo pois não temos que ficar checando a existência de dados. Aplicando a expressão idiomática, temos o resultado abaixo (para mais detalhes sobre essa expressão, veja na seção 2.4):

```
address = user_data['address'] || 'vazio'
address.strip
# "vazio"
```

Este caso é muito comum. Através do pouco usado método `#fetch`, temos uma outra maneira de retornar um valor *default*, muito mais legível e sem problemas com confusão de precedência de parâmetros:

```
address = user_data.fetch('address', 'vazio')
address.strip
# vazio
```

Notação para métodos

A comunidade Ruby adotou duas notações importantes quando se tratam de documentação de métodos. Métodos começando com # indicam que são aplicados à instâncias daquela classe, por exemplo `Hash#fetch`. Métodos começando com :: ou . indicam métodos de classe, por exemplo `Time::now`.

O Ruby tem um comportamento às vezes indesejável de retornar sempre `nil` quando a chave não existe:

```
user_data = {
  'email' => 'cicrano@example.com',
  'full_name' => 'Cicrano'
}

user_data['address']
# nil
```

Este caso é o que chamamos de falha silenciosa, ou seja, o código falhou, porém, como não há erro, a execução do código segue adiante, ficando difícil perceber quando há bugs. Dessa forma, quando fizer sentido tornar essa falha mais aparente, podemos usar o `#fetch` sem fallback, disparando uma exceção `KeyError` quando a chave não existe:

```
user_data = {
  'email' => 'cicrano@example.com',
  'full_name' => 'Cicrano'
}

user_data.fetch('address')
# KeyError: key not found: "address"
```

Símbolos

Símbolos são strings especiais. Símbolos são usados internamente pelo interpretador MRI para localizar o método a ser executado em um objeto, portanto sua implementação é tal que a torna imutável e única na instância do interpretador Ruby. Ou seja, uma vez um símbolo mencionado e criado, ele vai existir por todo o período de vida de execução do interpretador e nunca vai ser coletado pelo *garbage collector*:

```
a = "123"
b = "123"

a.object_id
# => 7022...7060
b.object_id
# => 7022...1360

a = :hello
b = :hello
```

```
a.object_id
# => 456328
b.object_id
# => 456328
```

> **O que é Garbage collector?**
>
> O *garbage-collector* é um mecanismo importante e complexo que faz parte do interpretador do Ruby. Ele é capaz de rastrear todos os recursos usados e não usados no sistema, de forma a liberar memória não utilizada ou alocar mais memória para seu programa quando necessário.

Em termos práticos, no Ruby 1.9.3, versão que usamos, isso não significa muita coisa. Porém, símbolos possuem outro valor: são usados no lugar de strings para serem identificadores especiais quando estamos nos referindo a métodos:

```
"string".respond_to? :upcase
# true
```

Também acontece para chaves de Hash, com uma pequena diferença:

- Símbolos devem ser usados quando nos tratamos de metadado e não dado em si. O que isso quer dizer? Usamos símbolos quando estamos descrevendo o tipo do dado e não dando um valor possível;

- Strings devem ser usadas quando a chave é um valor e não um descritor de dados.

Exemplificando as duas regras:

```
# Exemplo 1 - usando símbolos
{
  :name => 'Fulano',
  :email => 'fulano@example.com',
}

# Exemplo 2 - usando ambos
{
  :people => {
    'Fulano' => {
```

```
      :email => 'fulano@example.com'
    }
  }
}
```

Podemos facilmente converter strings em símbolos e vice-versa:

```
"um_simbolo".to_sym
# :um_simbolo

:um_simbolo.to_s
# "um_simbolo"

"E se a palavra for muito grande?".to_sym
# :"E se a palavra for muito grande?"

# Ainda é possível usar a notação de string para fazer interpolação!
method = 'flatten'
:"#{method}!"
# :flatten!
```

Ranges

Já vimos Ranges algumas vezes em outros exemplos. Range é um tipo interessante e simples. Um objeto Range possui um início, um fim e dois ou três pontos entre eles:

```
1..5  # Números inteiros entre 1 a 5, com o 5 inclusive
1...5 # Números inteiros entre 1 a 4, o 5 fica de fora

'a'..'e'   # Letras entre 'a' e 'e'
'a'...'e'  # Letras entre 'a' e 'd', o 'e' fica de fora
```

A maior utilidade de Ranges é testar se um valor está em um intervalo:

```
valid_years = 1920..2010
valid_years.include? 1998 # true
valid_years.include? 1889 # false
```

Há um detalhe importante sobre Ranges. Eles podem ser *discretos* ou *contínuos*. Por exemplo, ranges com floats são contínuos, ou seja, não existe um número finito

de valores inclusos dentro deste Range. Já um range discreto, existe um número finito e enumerável de elementos.

Isso se reflete em alguns métodos, inclusive o bastante útil `#to_a`:

```
years = 2000..2012
years.to_a
# [2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009,
#   2010, 2011, 2012]

length = 1.0..5.0
length.to_a
# TypeError: can't iterate from Float
```

true, false e nil

Diferente de outras linguagens, Ruby não possui um tipo "booleano". Tudo em Ruby é um objeto, e `true`, `false` e `nil` não fogem desta regra. Portanto, `true` é uma instância *singleton* (ou seja, apenas uma única instância dessa classe irá existir em todo o ciclo de vida de uma aplicação) da classe `TrueClass`, `false`, da `FalseClass` e `nil` da `NilClass`. Dessa forma, estes valores não são comparáveis com 0 e 1, como é comum em outras linguagens.

```
1 == 1
# true

1 > 1
# false

if "object"
  puts "Objetos em geral resultam em 'true'"
end
# Objetos em geral resultam em 'true'

if 0
  puts "0 é um objeto, portanto, true!"
end
# 0 é um objeto, portanto, true!

puts "nil é false em if" if nil
puts "falso é... falso." if false
```

Para verificar se um objeto é `nil`, é possível testar usando o método `#nil?`, implementado em todos os objetos:

```
a = nil
a.nil? # true

b = 1
b.nil? # false
```

2.4 Fluxos e laços

if

Como toda linguagem de programação imperativa, temos controles de fluxos inerente na sintaxe.

```
a = "0"
if a == "0"
  puts "É true!"
else
  puts "É falso :("
end

# É true!
# => nil
```

Uma coisa interessante que aconteceu no exemplo de código acima é que o IRB nos exibiu não uma, mas **duas** saídas. A primeira é a impressão do texto na tela, papel da função `puts`. Porém, como no primeiro trecho, temos o resultado com o símbolo =>, denotando o que o bloco de código em si retornou. Isso dá-se ao fato de que todo bloco de código Ruby retorna alguma coisa:

```
a = "0"
b = if a == "0"
      1
    else
      2
    end
# => 1
```

```
puts b # 1
puts a # "0"
```

Outro exemplo:

```
a = 0

if a == 0
  10
else
  20
end * 100
# => 1000
```

Note que, em Ruby, blocos de `if` terminam sempre com o `end`, com o `else` sendo opcional. Existe outra maneira de se fazer `if` no Ruby que é muito útil: o `if` como modificador, herança da linguagem Perl:

```
puts message if message # Imprime message se ela estiver definida
```

O `if` como modificador é muito conveniente e, dependendo de como for usado, torna o código bastante legível. Portanto é encorajado para se fazer trechos curtos (ou seja, uma linha) de código.

Para compor cláusulas em um `if`, podemos usar os operadores booleanos:

- `and` - Clássico "e", ou seja, **sempre** vai avaliar as duas expressões na direita e na esquerda e verificar se ambas as expressões são verdadeiras;
- `or` - Clássico "ou", ou seja, **sempre** vai avaliar as duas expressões na direita e na esquerda e verificar se pelo menos uma delas é verdadeira;
- `&&` - "E" lógico, é parecido com o `and`, porém possui a característica de "curto-circuito", ou seja, se a expressão da esquerda for **falsa**, a expressão da direita não vai ser avaliada;
- `||` - "Ou" lógico, parecido com o `or`, porém possui característica de "curto-circuito", ou seja, se a expressão da esquerda for **verdadeira**, a expressão da direita não é executada;
- `!` - O "not", ou seja, inverte `true` em `false` e vice-versa.

Exemplos:

```
age = 10
parents_are_together = true

puts "Não pode beber" if age < 18
puts "Pode votar, mas não beber" if age < 18 and age >= 16
puts "Pode votar, mas não beber" if age < 18 && age >= 16

puts "Pode ver o show" if age > 18 or parents_are_together
puts "Pode ver o show" if age > 18 || parents_are_together

puts "Pode ir pra balada" if !parents_are_together
```

Booleanos e valores *truthy* e *falsy*

"Truthy" e "falsy" são maneiras de explicar o comportamento de tipos (inteiro, string, etc.) em lógica de controle, tal como o `if`. Ou seja, se um valor é "truthy", o `if` é executado e, se "falsy", não.

No Ruby, apenas `false` e `nil` são "falsy", todos os outros tipos são "truthy", mesmo que o inteiro seja `0` (zero) ou se a string for "" (vazia). Tome bastante cuidado!

Atenção com precedência de operadores booleanos!

Os operadores booleanos são utilizados não somente para a composição de `if`, e portanto os operadores `or` e `||` possuem precedência diferentes (e o mesmo acontece para o "e"), tendo `or` e `and` menor precedência. Veja os exemplos abaixo:

```
do_something = "123"
do_other_stuff = "abc"

# Observemos que and e && retornam o valor da última sentença
# avaliada
do_something and do_other_stuff
# abc

do_something && do_other_stuff
# abc
```

```
# No ||, do_other_stuff não é avaliado por ser operador
# de curto circuito!
do_something || do_other_stuff
# 123

# Caso 1: and
if var = do_something and do_other_stuff
  puts "var is #{var}"
end
# var is 123

# Caso 2: &&
if var = do_something && do_other_stuff
  puts "var is #{var}"
end
# var is abc
```

Nestes exemplos, é possível ver que há uma confusão. No primeiro caso a associação (`var =`) tem maior precedência e é executado antes do `and`, ou seja, a execução é a mesma que `(var = do_something) and do_other_stuff`. Já no segundo caso, o `&&` é executado antes, portanto temos o mesmo que `var = (do_something && do_other_stuff)`. Fica muito difícil de ler, portanto a recomendação é sempre usar o operador `&&` e `||` e parênteses para deixar o código mais claro, ou ainda melhor, fazer a associação fora do `if`.

elsif

É possível ainda usar o `elsif`, uma combinação de `else` com `if`:

```
a = 1
# => 1

if a == 0
  puts "A é 0"
elsif a == 1
  puts "A é 1"
else
  puts "A não é nenhum"
end
# A is 1
```

unless

Também herdado do Perl é o `unless`, que basicamente inverte o comportamento do `if`, ou seja, o bloco associado é executado se a expressão associada retorna `false`:

```
a = 0
# => 0

# Uso comum, sem else
unless a == 0
  puts "A não é zero"
end
# => nil

# Uso com else
unless a == 0
  puts "A não é zero"
else
  puts "A é zero, na verdade!"
end
# A é zero, na verdade
# => nil

# Uso do unless como posfixo
puts "A é zero" unless a > 0
# A é zero
# => nil
```

> **Dica: Cuidados com o uso do `unless`**
>
> O `unless` pode ser muito confuso para entender caso você tenha múltiplas cláusulas na expressão associada, já que ele inverte a lógica do `if` que já é de costume.
>
> Outro problema do `unless` é com o `else`, por exatamente o mesmo motivo. É preferível, neste caso, inverter os blocos e usar `if` diretamente, que é muito mais legível e sem "pegadinhas".

O operador ternário `?` também pode ser usado, conhecido de linguagens como Java e C:

```
a = 0
# => 0

a == 0 ? puts("É zero!") : puts("Não é zero")
# É zero!
# => nil
```

O operador ternário `?` recebe, do seu lado esquerdo a expressão a ser avaliada. No seu lado direito, a primeira parte é para o caso da expressão ser `true` e o outro lado, caso seja `false`.

Use com cautela, pois, nesse caso por exemplo, é necessário explicitar os parênteses devido a confusão com precedência de operadores. O `?` deve ser usado com pequenas linhas de código para não prejudicar a legibilidade.

case

O `case` é usado quando há muitas condições a serem checadas, evitando muitos `else` e `elsif`:

```
case
when a > 0
  puts "A é maior que 0"
when a < 0
  puts "A é menor que 0"
when a == 0
  puts "A é 0"
else
  puts "Quebramos a matemática!"
end

# A é 0
# => nil
```

Cascade em Ruby

Em linguagens como C e JavaScript, uma vez que o interpretador/compilador encontra um caso verdadeiro, todo o código restante é executado, independente se os resultados dos outros *cases* sejam `false`. Esse fenômeno é chamado de *cascade*:

```
// Cascade em JavaScript:
var a = 1;
switch(a) {
  case 0:
    console.log("0");
  case 1:
    console.log("1");
  case 2:
    console.log("2");
}

// 1
// 2
```

Para resolver esse problema, é necessário executar um `break`:

```
// Cascade em JavaScript:
var a = 1;
switch(a) {
  case 0:
    console.log("0");
    break;
  case 1:
    console.log("1");
    break;
  case 2:
    console.log("2");
    break;
}

// 1
```

Esse fenômeno **não** acontece com Ruby, portanto não é necessário temperar o `case` com "breaks".

O case é especial. Para alguns tipos de objetos, ele pode reagir diferente, veja os exemplos a seguir:

```
case number_of_kills
when 0..10
  puts ""
when 11..20
  puts "Monster kill!"
when 21..40
  puts "Rampage!"
when 41..70
  puts "DOMINATING!"
else
  puts "GODLIKE!"
end
```

Neste exemplo, verificamos a que intervalo pertence a variável number_of_kills com o uso de Range (veremos mais detalhes sobre Ranges na seção 2.3). Vejamos agora um exemplo com strings:

```
case input_date
when /\d{4}-\d{2}-\d{2}/
  puts "O formato é yyyy-mm-dd"
when /\d{2}-\d{2}-\d{4}/
  puts "O formato é dd-mm-yyyy"
when /\d{2}-\d{2}-\d{2}/
  puts "O formato é dd-mm-yy"
else
  puts "Não sei que formato é"
end
```

Neste caso, o bloco associado ao when irá ser executado quando a string bater com expressão regular.

Case statement operator

Não é escopo deste livro ensinar internamente como o Ruby verifica isso, mas se você tiver interesse em saber como isto é feito, busque pelo método "===" ou "case statement operator".

while e until

`while` e `until` são conhecidos de qualquer linguagem imperativa. O `while` executa um bloco associado de código enquanto a condição estiver satisfeita:

```
a = [1, 2, 3]

while a.length > 0 do
  puts "Bye bye, #{a.pop}"
end
# Bye bye, 3
# Bye bye, 2
# Bye bye, 1
# => nil
```

O `until` ("até que" em inglês) faz justamente o contrário. Ele executa um bloco associado de código até que a situação se faça verdadeira:

```
a = [1, 2, 3]

until a.empty? do
  puts "Bye bye, #{a.pop}"
end
# Bye bye, 3
# Bye bye, 2
# Bye bye, 1
# => nil
```

> **Cuidados com `until`**
>
> O `until` possui o mesmo problema semântico do `unless` (seção 2.4), ou seja, podemos ter problemas ao pensar em dupla negativa. Dessa forma, recomenda-se usar apenas `while`, negando a expressão quando pertinente.

for ... in

O `for ... in` é uma maneira de iterar elementos de uma coleção, ou um objeto "iterável", ou seja, um objeto que seja um `Enumerator`. A estrutura do `for` é simples:

`for` *`variável`* `in` *`coleção`*

Vejamos um exemplo com Arrays, que são "iteráveis":

```
fruits = %w{pera uva maçã}

for fruit in fruits
  puts "Gosto de " + fruit
end
# Gosto de pera
# Gosto de uva
# Gosto de maçã
# => ["pera", "uva", "maçã"]
```

Hashes também são iteráveis, com uma pequena diferença:

```
frequencies = {'hello' => 10, 'world' => 20}

for word, frequency in frequencies
  puts "A frequência da palavra '#{word}' é #{frequency}"
end
# A frequência da palavra 'hello' é 10
# A frequência da palavra 'world' é 20
# => {"hello"=>10, "world"=>20}
```

Blocos

Blocos são estruturas singulares a Ruby, e são extremamente importantes, tanto é que merecem atenção especial. Blocos são trechos de código associados a um método que podem ser executados a qualquer momento por este método.

Este conceito é um pouco estranho para quem vem de linguagens como Java (apesar de que Java possui uma forma de se fazer blocos, com classes anônimas) e C, mas se você já ouviu falar de *lambdas*, está familiarizado com o conceito.

O exemplo mais clássico de blocos é o uso de iteradores:

```
fruits = %w{pera uva maçã}

fruits.each do |fruit|
  puts "Gosto de " + fruit
end
# Gosto de pera
# Gosto de uva
```

```
# Gosto de maçã
# => ["pera", "uva", "maçã"]
```

Explicando o exemplo anterior, o método `#each` pega cada valor dentro de sua coleção e repassa para o bloco associado, através da variável `fruit`.

Outro método que usa bloco bastante útil é o `#map`, de Hashes e Arrays, que retorna um novo Array contendo como elementos o resultado devolvido a cada iteração:

```
numbers = [1, 2, 3, 4]

squared_numbers = numbers.map { |number| number * number }
squared_numbers  # [1, 4, 9, 16]
```

Notação para blocos

Como você pôde perceber, foram usadas duas notações distintas nos exemplos anteriores, a notação com `do ... end` e a notação com chaves. Ambas funcionam da mesma maneira. Porém, a comunidade Ruby adotou a seguinte regra:

- Para blocos curtos, de apenas uma linha, adota-se as chaves;
- Para blocos longos, de duas ou mais linhas, adota-se o `do ... end`.

A grande utilidade de blocos é que existem inúmeras APIs que exigem operações antes e depois de serem utilizadas. Com o uso de blocos, este procedimento fica transparente, como é possível observar no próximo exemplo:

```
# Exemplo tradicional
file = File.new('file.txt', 'w')
file.puts "Escrevendo no arquivo"
file.close

# Ao invés de termos que nos preocupar com a abertura e o fechamento do
# arquivo, a própria API faz isso antes e depois do bloco.
```

```
File.open('another_file.txt', 'w') do |file|
  file.puts "Escrevendo no arquivo com blocos!"
end
```

É possível também passar e receber mais de um argumento para blocos. O funcionamento é bem o mesmo de argumentos para funções, vejamos o exemplo abaixo com Hashes:

```
a = {:a => 1, :b => 2, :c => 3}

a.each do |key, value|
  puts "A chave #{key} possui valor #{value}"
end

# A chave a possui valor 1
# A chave b possui valor 2
# A chave c possui valor 3
```

Escopo de variáveis

Variáveis declaradas no mesmo escopo em que o bloco se encontra são acessíveis de dentro deste bloco.

```
a = %w{a b c d e}

counter = 0
a.each { |val| counter += 1 }

puts "O valor do contador é: #{counter}" # O valor do contador é: 5
```

Se usarmos um nome de variável para o parâmetro de bloco que é o mesmo que uma variável externa ao bloco, a variável externa deixará de ser acessível dentro do bloco, mas terá seu valor mantido:

```
a = %w{a b c}
i = 5

a.each do |i|
  puts i
end # a; b; c

puts i # 5
```

Atenção ao escopo!

O comportamento do exemplo anterior é restrito à variáveis em parâmetros. Isso significa que variáveis externas ao bloco são modificáveis dentro deste mesmo bloco, conforme o primeiro exemplo. O resultado disso pode ser inesperado, já que o bloco pode estar alterando valores de outras variáveis.

As variáveis declaradas dentro do bloco são criadas a cada vez que o bloco é chamado, ou seja, essas variáveis não existirão caso o bloco seja chamado novamente e serão criadas novamente:

```
a = %w{w o r d}

a.each do |letter|
  word ||= ""
  word << letter
  puts word
end
# w
# o
# r
# d
```

Existem situações em que nós não ligamos para um dos valores passados para o bloco, mas ainda assim precisamos respeitar o número de parâmetros. Podemos assim usar o _, que irá "absorver" esse parâmetro sem ter a necessidade de ter que declarar uma nova variável:

```
a = {:a => 1, :b => 2, :c => 3}

a.each do |_, value|
  puts value
end # 1; 2; 3
```

É recomendado sempre usar o _ ao invés de simplesmente omitir os valores. Algumas vezes é possível simplesmente omitir, porém existem métodos cujo comportamento muda de acordo com o número de parâmetros passados (conhecido como

arity). Isso acontece porque é possível verificar quantos parâmetros o bloco pode receber:

```
a = {:a => 1, :b => 2, :c => 3}

a.each do |key|
  puts key
end
# a
# 1
# b
# 2
# c
# 3
```

Controle de fluxo em blocos

No caso de uso de blocos como iteradores, é possível controlar o fluxo com algumas expressões. Existem várias que raramente são usadas. As principais são as seguintes:

- `break` - Pára o iterador completamente no momento em que for chamado;
- `next` - Passa para o próximo elemento;

Vejamos alguns exemplos:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
numbers.each do |number|
  next if number.odd?

  puts number
end # 2; 4; 6; 8; 10

numbers.each do |number|
  break if number > 5

  puts number
end # 1; 2; 3; 4; 5
```

2.5 Funções, blocos, lambdas e closure

Em Ruby, declarações de funções e métodos são assim:

```
def print_text(text)
  puts text
end

# É possível omitir os parênteses
def print_text text, more_text
  puts text + more_text
end
```

Retorno de métodos e funções

Diferente da maioria das linguagens, em Ruby não é necessário colocar um `return` para retornar um valor, porque o último código executado de um método será o objeto retornado:

```
def double_that(number)
  var = "isso não é relevante"
  number * number
end

puts double_that(10) # 100
```

Lembrando que a maioria dos blocos de código Ruby retornam valor, tal como o próprio `if` (seção 2.4). Portanto, é possível construir funções da seguinte maneira:

```
def am_i_rich?(cash)
  if cash > 1_000_000
    "sim"
  else
    "não"
  end
end

puts am_i_rich?(100) # "não"
```

Às vezes, porém, é interessante retornar em um ponto específico da função. Dessa forma, usamos o `return`:

```
def factorial(n)
  return 1 if n == 1

  n * factorial(n-1)
```

```
end

factorial(1) # 1
factorial(3) # 6
```

No exemplo anterior, no caso em que `n` é 1, a função retorna `1`, sem executar o trecho na linha 4.

> **`return` E BLOCOS**
>
> Cuidado com o uso de `return` dentro de blocos. Primeiro porque não existe um comportamento intuitivo para o `return` de dentro de blocos. Ele deve encerrar o *loop*, tal como o `break`? Ele deve ceder o fluxo de volta para o método original, tal como o `next`? Nenhum dos dois. Ele de fato não tem nenhum comportamento especial com blocos, ou seja, o `return` termina a execução do método/função a qual o bloco está associado.
>
> A segunda razão para não fazer isso é que, se houver um código importante que precisa ser executado posterior ao bloco, como no caso do `File.open`, ele deixará de ser executado.

alias

É possível criar múltiplos nomes para uma função via a expressão `alias`:

```
def factorial(n)
  return 1 if n == 1
  n * factorial(n-1)
end

alias fac factorial

fac(5) # 120
```

Esta possibilidade pode parecer meio inútil a princípio, mas, em diversas situações, torna a linguagem mais expressiva e natural, ou seja, a leitura de código Ruby tende a ser bem próxima do inglês. Dessa forma, criam-se *aliases* para funções de forma a melhorar a legibilidade do código produzido. O exemplo a seguir é usado no próprio Rails:

```
require 'active_support/all'

1.hour.ago    # 2012-04-30 20:15:26 -0300

# Leitura péssima, mas funciona
2.hour.ago    # 2012-04-30 19:15:26 -0300

# Agora faz sentido!
2.hours.ago   # 2012-04-30 19:15:26 -0300
```

Parâmetros default

É possível criar funções e métodos com parâmetros padrão usando a seguinte sintaxe:

```
def truncate(string, length=20)
  string[0,length-3] + "..."
end

puts truncate("Truncando essa linha longa")     # Truncando essa li...
puts truncate("Truncando essa linha longa", 10) # Truncan...
```

Números de argumentos variáveis

Em Ruby podemos criar métodos que aceitam um número variado de argumentos. Conseguimos isso através do uso do operador *splat*:

```
def sum(*values)
  puts values.inspect

  values.reduce { |sum, value| sum + value }
end

sum(1)              # [1]; 1
sum(1, 1)           # [1,1]; 2
sum(1, 2, 3, 4, 10) # [1, 2, 3, 4, 10]; 20
```

É possível destacar alguns parâmetros caso isso seja interessante:

```
def sum(first, *values)
  puts values.inspect
```

```
  values.reduce(first) { |sum, value| sum + value }
end

sum(1)              # []; 1
sum(1, 1)           # [1]; 2
sum(1, 2, 3, 4, 10) # [2, 3, 4, 10]; 20
```

Hashes para parâmetros nomeados

Alguns métodos podem precisar receber muitos parâmetros. Agora, imagine uma chamada de método dessa forma: `process(1, 10, friends, Date.today, 3.days.from_now)`. Fica bem difícil lembrar o que cada parâmetro significa e pior, qual é a ordem desses parâmetros. Ou mais além, no Rails, por exemplo, é possível passar algumas opções para mudar algum comportamento do método. Este tipo de problema é o chamado **acoplamento por posição**, ou seja, o código torna-se complicado devido à grande dependência da posição dos parâmetros.

Por isso, é bem comum usar *hashes* como parâmetros para tornar a chamada de método mais legível e evitar que o programador fique consultando manuais a todo momento:

```
search_flights(:from => 'SAO', :to => 'NYC')

search_flights(:from      => 'SAO',
               :to        => 'NYC',
               :max_stops => 3,
               :class     => :first)
```

Resumindo, os principais usos para *hashes* como parâmetros são:

- Independência de ordem de parâmetros, reduzindo acoplamento por posição;
- Quando há muitos parâmetros;
- Quando há parâmetros opcionais;
- Quando é possível alterar o comportamento através de opções.

Saiba mais: acoplamento por posição

Para entender melhor sobre este tipo de dependência e acoplamento, recomendo a leitura do artigo "Connascence as a Software Design Metric [4]" do Gregory Brown e também ver a palestra "Building Blocks of Modularity [10]", do Jim Weirich.

Nesses casos, é bastante comum métodos serem construídos da seguinte maneira:

```
# Se nada for passado, todos os parâmetros serão os default
def search_flights(options={})
  from         = options.fetch(:from, 'SAO')
  to           = options.fetch(:to, '')
  max_stops    = options.fetch(:max_stops, 9999)
  flight_class = options.fetch(:class, :any)

  # ...
end
```

Blocos associados a métodos

Vimos no final da seção 2.4 o funcionamento de blocos. Agora veremos como construir funções que possuem comportamento semelhante. O procedimento é bastante simples, basta usarmos duas expressões: `yield` e `block_given?`.

O `yield` passa o fluxo de execução do programa para o bloco associado ao método ou função. Usamos o `block_given?` para verificar se algum bloco foi passado de fato ao método. Observe os exemplos a seguir:

```
def announce_it
  puts "Legen..."
  yield
  puts "Dary!"
end

announce_it { puts "Wait for it!" }
# Legen...
# Wait for it!
# Dary!
```

```
announce_it
# Legen...
# LocalJumpError: no block given (yield)
```

Agora, usando o `block_given?`:

```
def announce_it
  puts "Legen..."

  # yield não executado caso não haja um bloco!
  yield if block_given?

  puts "Dary!"
end

announce_it { puts "Wait for it!" }
# Legen...
# Wait for it!
# Dary!

announce_it
# Legen...
# Dary!
```

Sintaxe alternativa para blocos

Existe uma forma de se usar blocos que não usa o par `yield`/`block_given?`. Essa forma é explicitar o bloco com um parâmetro da função através do operador `&`, que sempre deverá vir por último na lista de parâmetros:

```
def announce_it(name, &block)
  puts "Hey #{name}, it's gonna be... "
  puts "Legen..."

  # Se o bloco não for passado, block será nil
  block.call if block
  puts "Dary!"
end

announce_it("Ted")
announce_it("Ted") { puts "Wait for it..." }
```

Apesar de não recomendado, mesmo usando esta forma é possível usar `yield` e `block_given?`. A grande vantagem de usar esta forma é para repassar o bloco para outros métodos, sendo impossível com o `yield`:

```
def header(&block)
  puts "Começa..."

  # Precisamos transformar a variável
  # em bloco novamente, por isso colocamos o &
  [1,2,3].each(&block)

  puts "Termina!"
end

header { |num| puts num }
# Começa...
# 1
# 2
# 3
# Termina!
# => nil
```

> **Sintaxe de blocos**
>
> A sintaxe de blocos mais usada por desenvolvedores que trabalham com Rails é a mesma dos desenvolvedores do próprio Rails: explicitar os blocos no parâmetro com &. O motivo é simples: documentação, ou seja, fica claro para o leitor do código que aquele método pode receber um bloco.

procs

Uma coisa interessante na sintaxe usando o operador `&` (apelidado também de operador "pretzel") é que o bloco é transformado em um objeto da classe `Proc`, que responde ao método `#call`:

```
def block_it(&block)
  puts block.class
end
```

```
block_it {} # Proc
```

Procs são estruturas amplamente utilizadas em projetos Ruby por diversas razões, sendo a possibilidade de transformá-las em blocos é apenas uma delas. É importante ressaltar esse fato: blocos **não** são procs e vice-versa, mas são facilmente transformados um no outro usando o &. Vejamos alguns exemplos:

```
logger = proc { |x| puts "#{Time.now} -- #{x}" }

logger.call("Teste!") # 2012-05-08 15:52:58 -0300 -- Teste!
[1,2,3].each(&logger) # O proc vira bloco para o each com o &
# 2012-05-08 15:53:46 -0300 -- 1
# 2012-05-08 15:53:46 -0300 -- 2
# 2012-05-08 15:53:46 -0300 -- 3
```

Um uso extremamente útil desses conceitos é que símbolos implementam o método #to_proc, que é chamado pelo operador &. Este proc resultante chama o método do argumento cujo nome é o símbolo. Para ficar mais claro, vejamos os exemplos a seguir.

Primeiramente, vejamos como que funciona o #to_proc:

```
upcase_it = :upcase.to_proc

upcase_it.call('abcde') # ABCDE
upcase_it.call(123) # NoMethodError: undefined method
                    # `upcase' for 123:Fixnum
```

Ou seja, o proc gerado basicamente chama o método no objeto. Porém, usado com o &, podemos criar filtros de maneira simples e enxuta:

```
# Forma tradicional usando bloco
%w{pera uva jaca}.map { |fruit| fruit.upcase }
# ["PERA", "UVA", "JACA"]

# Usando o #to_proc
%w{pera uva jaca}.map(&:upcase)
# ["PERA", "UVA", "JACA"]
```

lambdas

Lambdas são bem parecidos com procs, e muitas vezes são usados de forma intercambiável. A forma de construir lambdas é bastante parecida com procs:

```
upcase_it = lambda { |x| x.upcase }
upcase_it.call("abc")  # ABC
# Arity é o número de parâmetros que o lambda aceita
upcase_it.arity # 1
```

Porém, lambdas possuem um atalho especial:

```
upcase_it = ->(x) { x.upcase }
upcase_it.call("abc") # ABC
```

Não existe uma recomendação para o uso das notações. Porém a primeira é mais utilizada apenas por razões históricas: a notação `->()` é uma introdução recente à linguagem. Fique à vontade para usar a que você mais gostar.

Diferenças entre procs e lambdas

Apesar de serem parecidos, procs e lambdas não são iguais em dois comportamentos.

Primeiro, vimos anteriormente que variáveis em excesso não causam nenhum problema com blocos e são simplesmente ignorados, e isso acontece com procs, mas não com lambdas:

```
show = proc { |x, y| puts "#{x}, #{y}" }

show.call(1)          # 1,
show.call(1, 2, 3)    # 1, 2

show = ->(x,y) { puts "#{x}, #{y}" }

show.call(1, 2)       # 1, 2
show.call(1)          # ArgumentError: wrong number of arguments (1 for 2)
show.call(1, 2, 3)    # ArgumentError: wrong number of arguments (3 for 2)
```

A segunda diferença é no uso do `return`. Dentro de blocos, quando usamos `return`, o que de fato é retornado é o método associado, já que blocos/procs não consideram o `return`. Em contrapartida, lambdas são mais próximos a métodos e funções e retornam apenas ao contexto a que pertencem:

```
def proc_stop
  puts "Cheguei..."
  proc { puts "Hey"; return; puts "Ho!" }.call
  puts "Saindo..."
end

proc_stop # Cheguei...; Hey

def lambda_stop
  puts "Cheguei..."
  lambda { puts "Hey"; return; puts "Ho!" }.call
  puts "Saindo..."
end

lambda_stop # Cheguei...; Hey; Saindo...
```

closures

"Closure", ou fechamento, é uma característica de funções que podem ser associadas a variáveis e podem ser invocadas a qualquer momento. Quando funções com essa característica são criadas, elas carregam em si não apenas o código a ser executado mas também referências à todas as variáveis existentes naquele escopo.

Em Ruby, lambdas e procs são closures, portanto carregam em si referências às variáveis em seu escopo. Vejamos um exemplo:

```
def create_lambda
  value = 10

  -> { value += 1; puts value }
end

l = create_lambda
l.call # 11
l.call # 12
```

O que acontece no exemplo anterior é que, quando o lambda é criado, a variável `value` está em seu escopo e portanto o lambda possui uma referência à `value`. Mesmo quando a função `create_lambda` termina de executar, o lambda `l` ainda consegue acessar a variável `value` pois ainda possui uma referência. Note o exemplo a seguir, criando duas lambdas:

```
def create_lambda
  value = 10
  -> { value += 1; puts value }
end

first_lambda = create_lambda
next_lambda  = create_lambda

first_lambda.call # 11
next_lambda.call  # 11 - "value" é outra variável neste escopo

first_lambda.call # 12
first_lambda.call # 13
```

Quando executamos `create_lambda` pela segunda vez, uma nova variável `value` foi criada e associada ao escopo de `next_lambda`. Dessa forma, `next_lambda` e `first_lambda` referenciam-se à diferentes `value`, criadas em momentos diferentes.

Agora vamos modificar um pouco o código. Vamos criar duas lambdas dentro de um mesmo escopo:

```
def create_lambdas
  value = 10

  first = -> { value += 1; puts value }
  last  = -> { value += 1; puts value }

  [first, last]
end

first_lambda, last_lambda = create_lambdas

first_lambda.call # 11
last_lambda.call  # 12 - "value" é a mesma variável

first_lambda.call # 13
last_lambda.call # 14
```

É possível observar no exemplo anterior que ambos `first_lambda` e `last_lambda` carregam o mesmo escopo, ou seja, compartilham a mesma referência à variável `value`.

DICA: CUIDADO COM ESCOPO DE CLOSURES

Há situações que não é possível saber que momento e nem em que ordem que lambdas e procs são executados pois são propagados à outras partes do código. Dessa forma, compartilhar variáveis via closures pode resultar em comportamento difícil de prever e bugs bem complicados de serem encontrados. Por isso, preste bastante atenção nas variáveis sendo compartilhadas via closures.

2.6 Classes e módulos

Ruby é uma linguagem orientada a objetos, e portanto, não poderia faltar uma sintaxe para declaração de classes, que no caso, é com a palavra-chave `class`:

```
class Purchase
end
```

Veja que, para a declaração de classes, usa-se o padrão `CamelCase`, porém começando com uma letra maiúscula. Isso deve-se ao fato que qualquer nome, seja de variável ou de classe, que começa com uma letra maiúscula é uma constante.

A classe `Purchase` não tem nada de interessante. Vamos torná-la um pouco melhor.

Construtores

```
class Purchase
  def initialize(value)
    @value = value
  end
end

payment = Purchase.new(100.00)
payment # #<Purchase:0x007f97e14e61e8 @value=100.0>
```

No exemplo anterior acontecem algumas coisas interessantes. Primeiramente, criamos um método de *instância* chamado `#initialize`. Métodos de instância são métodos que podem ser executados em instâncias da classe `Purchase`.

Mas esse método não é qualquer método. Ele é um *construtor*, ou seja, ele é chamado quando um novo objeto é criado. Você pode observar porém que não chamamos o método `#initialize` diretamente, mas o `.new`. Por que? O Ruby implementa no método de classe `.new` (ou seja, método relacionado diretamente à classe e não ao objeto) as mágicas internas necessárias para se criar um novo objeto e, em seguida, repassa tudo que recebeu para o método `#initialize` e retorna a nova instância do objeto (independente do último valor).

Essa nova instância do objeto criada é chamada também de `self`. Dessa forma, usando variáveis com "arroba" (`@`) é possível associar valores apenas àquela instância. Por essa razão, elas são chamadas de variáveis de instância.

Declaração de métodos

Declarar métodos é bem simples, basta seguir o exemplo do `#initialize`:

```
class Purchase
  def initialize(value, shipping)
    @value = value
    @shipping_cost = shipping
  end

  def total_cost
    @value + @shipping_cost
  end
end

purchase = Purchase.new(100.00, 9.50)
purchase.total_cost # 109.5
```

Accessors

"Accessors" são os famosos "getters" e "setters" do Ruby. Ou seja, são métodos de leitura e escrita. Uma coisa interessante do Ruby é que chamada de métodos, em muitas situações, não precisam de parênteses. Por essa razão, um método de leitura de variáveis, por exemplo, se parece como um simples acesso a uma variável pública. Não há necessidade também de nenhuma forma de prefixo, tal como `getShippingCost`, como faríamos em outras linguagens.

```
class Purchase
  def initialize(value, shipping)
```

```
    @value = value
    @shipping_cost = shipping
  end

  def shipping_cost
    @shipping_cost
  end
end

purchase = Purchase.new(100.00, 9.50)
purchase.shipping_cost # 9.5
```

O "setter", ou método de escrita, é bem mais interessante. Como já vimos anteriormente, é possível escrever métodos com símbolos como `?` e `!`. Mas também, é possível escrever métodos terminando com = para criar expressões de associação:

```
class Purchase
  def initialize(value, shipping)
    @value = value
    @shipping_cost = shipping
  end

  def shipping_cost=(new_shipping_cost)
    @shipping_cost = new_shipping_cost
  end

  def shipping_cost
    @shipping_cost
  end
end

purchase = Purchase.new(100.00, 9.50)
purchase.shipping_cost     # 9.5
purchase.shipping_cost = 3 # é possível colocar espaço antes do '='

purchase.shipping_cost     # 3
```

Cuidado com métodos de leitura e escrita

Métodos de leitura e escrita são para estes propósitos. Então tome cuidado com a sua implementação destes métodos. Se eles fizerem muito mais do que escrever e ler variáveis de instância, talvez não fique óbvio para quem esteja lendo o código o que o método está fazendo. Não surpreenda negativamente os leitores do seu código!

Para facilitar este tipo de construção, existe uma *class macro* que já cria esses métodos para você. Basta chamar `attr_accessor` no corpo da classe, passando como parâmetro uma lista de símbolos que representa o atributo a ser criado.

```
class Purchase
  attr_accessor :shipping_cost
end

purchase = Purchase.new
purchase.shipping_cost = 10.0
purchase.shipping_cost # 10.0
```

O que é class macro?

Uma *class macro* é uma chamada de método no nível de classe e o resultado dessa chamada é a alteração da classe, adicionando comportamento e tornando algum comportamento mais conveniente ao programador. Por exemplo, a macro `attr_accessor` gera métodos de leitura e escrita para um certo atributo.

A macro `attr_accessor` já cria o método de leitura e escrita para você. É possível porém ter apenas o método de leitura via a macro `attr_reader` e o método de escrita via `attr_writer`.

Cuidado com acessores de escrita

Existe um problema muito comum que os *accessors* de escrita geram e que é bastante complicado de depurar, portanto preste atenção! Imagine a seguinte situação:

```ruby
class Purchase
  attr_accessor :shipping_cost, :weight, :distance

  def calculate_shipping_cost
    shipping_cost = distance * weight
  end
end

purchase = Purchase.new
purchase.weight = 0.5
purchase.distance = 200

purchase.calculate_shipping_cost

purchase.shipping_cost # Qual é o valor?
```

Se você percebeu o mistério, já deve estar duvidando que `shipping_cost` seja 100. De fato, o valor não é 100 e sim `nil`. Mas qual a razão? Na linha 5, o Ruby simplesmente interpreta a associação como a criação de uma variável local, ao invés de procurar uma chamada de método de escrita. Portanto, é imprescindível que, quando usarmos *acessors* de *escrita* sejam acompanhados de `self`. Observe que *acessors* de leitura não necessitam dessa correção.

```ruby
class Purchase
  attr_accessor :shipping_cost, :weight, :distance

  def calculate_shipping_cost
    self.shipping_cost = distance * weight
  end
end

purchase = Purchase.new
purchase.weight = 0.5
purchase.distance = 200

purchase.calculate_shipping_cost

purchase.shipping_cost # Agora sim, 100.0!
```

Declarando métodos de classe

Não vimos anteriormente mas é possível declarar método em *qualquer* objeto, não apenas uma classe. Vejamos o seguinte exemplo:

```
a = "123"
def a.scream
  puts "AAAAAAAARGH!"
end

a.scream # "AAAAAAAARGH!"

b = "abc"
b.scream # NoMethodError: undefined method `scream' for "abc":String
```

Usamos essa mesma ideia para declarar métodos de classe:

```
1  class Purchase
2    attr_accessor :shipping_cost, :weight, :distance
3
4    def Purchase.build_free_shipping
5      purchase = new
6      purchase.shipping_cost = 0
7
8      purchase
9    end
10 end
11
12 Purchase.build_free_shipping # #<Purchase:0x0...0 @shipping_cost=0>
```

Métodos de classe executam no escopo de classe (ou seja, `self` é a própria classe), então métodos como o construtor (como na linha 5 no exemplo anterior) são acessíveis sem ter que especificar a própria classe.

Porém, podemos melhorar isso. Dentro do bloco `class`, `self` é a própria classe. Por isso, ao invés de termos que repetir o nome da classe, podemos usar simplesmente `self`:

```
class Purchase
  puts self # Purchase

  def self.build_free_shipping
    #...
```

```
  end
end
```

Herança

Impossível falar sobre uma linguagem orientada a objetos sem mencionar herança. Herança nada mais é do que a especialização de uma classe, muitas vezes mudando apenas alguns comportamentos pontuais e ainda mantendo algumas características fundamentais, como nome de método.

```
class Shipping
  attr_accessor :distance, :dimension

  def cost
    cubed_weight_factor = 16.7

    distance * dimension/1000 * cubed_weight_factor
  end
end

class UltraShipping < Shipping
  def cost
    super + (distance) * 0.07
  end
end

shipping = UltraShipping.new
shipping.distance = 200
shipping.dimension = 1.2

shipping.cost # 18.00800...
```

No exemplo anterior, é possível perceber que, independente se usarmos `Shipping` ou `UltraShipping`, o código vai funcionar pois ambas as classes respondem aos mesmos métodos.

Este comportamento é possível através de duas características importantes da linguagem. A primeira é a declaração de herança, usando `<`. Em seguida, vimos também como é possível chamar métodos de mesmo nome da super classe usando o `super`. O `super` é bastante importante para dizer ao Ruby que ele deve buscar na Árvore Genealógica da classe (classe mãe, classe avó e assim vai) o método a ser executado.

Lembrando também que construtores (`new`) nada mais do que são métodos, então também são herdados!

Visibilidade de métodos

É **muito** importante quebrar seu código em vários métodos. Porém, muitas vezes eles servem apenas para uso interno dentro das classes, portanto é importante declará-lo como `private`, ou seja, métodos apenas acessíveis internamente à classe:

```
class Invoice
  attr_accessor :service

  def total_price
    service + tax
  end

  private

  def tax
    service * 0.008
  end
end

invoice = Invoice.new
invoice.service = 1000

invoice.total_price # 1008.00
invoice.tax #  NoMethodError: private method `tax'
            #   called for #<Invoice:0x007fdf92810550 @service=1000>
```

Diferente de muitas linguagens em que `private` ou `public` é prefixado à declaração de todos os métodos, no Ruby funciona como uma chave que começa ligada no `public`. Quando você então escreve `private`, a chave é virada e todos os métodos então declarados são privados. Você pode voltar a chave para público chamando `public`.

Métodos privados também são acessíveis de classes filhas. Isso pode ser uma surpresa para você leitor, pois este comportamento não é comum. Dessa forma, não é necessário usar `protected` para que classes filhas tenham acesso a esses métodos. Fique atento a isso!

protected

O `protected` existe em Ruby e frequentemente é usado erroneamente devido ao seu comportamento em outras linguagens. A real utilidade do `protected` é praticamente nula e portanto deve ser evitada.

Mas para você, desenvolvedor curioso, fica a explicação do `protected`: o funcionamento do `protected` é quase o mesmo do `private`, contudo, é possível que um objeto de mesma classe ou subclasse chame o método (igual ao `friend`, do C++). Vejamos o exemplo a seguir:

```
class Person
  attr_accessor :name

  def befriend(people)
    people.each { |friend| friend.add_friend(self) }
  end

  protected

  def add_friend(friend)
    puts "#{name} diz: Olá meu novo amigo #{friend.name}!"
  end
end

joao = Person.new; joao.name = 'João'
pedro = Person.new; pedro.name = 'Pedro'
joaquim = Person.new; joaquim.name = 'Joaquim'

joao.befriend([pedro])
# Pedro diz: Olá meu novo amigo João!

joaquim.add_friend(joao)
# NoMethodError: protected method `add_friend' ...
```

Exercício para você leitor! Altere o `protected` para `private` e veja o que acontece. Depois, `public`.

Módulos

Módulos em Ruby são basicamente agrupadores de métodos, constantes, classes e variáveis. O uso é quase o mesmo de classes, porém não podemos criar uma

instância de um módulo e usamos a palavra chave `module`, ao invés de `class`. Outra diferença é que não existe hierarquia de módulos, ou seja, não faz sentido um módulo "herdar" de outro.

Módulo são usados de duas maneiras, usados como *namespaces*, ou seja, uma boa maneira de agrupar classes, constantes e métodos quando pertinente (por exemplo, agrupar todas as classes que fazem parte do meio de pagamento de um site) ou como *mixins*, uma forma de adicionar um comportamento comum a qualquer classe. Veremos mais exemplos sobre essa importante funcionalidade.

Módulos como namespaces

Quando um sistema se torna grande, é importante agrupar diversas classes que fazem parte de um componente em um *namespace*. Esta funcionalidade é de extrema importância, por exemplo, quando temos diversas classes com o mesmo nome, seja por um código que escrevemos ou por alguma biblioteca que usamos. Se não criarmos *namespaces*, comportamentos estranhos podem acontecer.

Para construir um módulo, basta usar uma sintaxe muito parecida com a de classes:

```
module Payment
  class Purchase
  end
end

Purchase.new # NameError: uninitialized constant Purchase
Payment::Purchase.new # => #<Payment::Purchase:0x007fac81d0b0f8>
```

No exemplo anterior podemos notar duas importantes características de módulos:

- Módulos devem também ser nomeados como constantes, ou seja, `CamelCase` com a primeira letra em maiúscula;

- Quando declaramos uma constante ou classe dentro de um módulo, é necessário explicitar o escopo via `::`.

O uso do `::` pode ser omitido no caso de estarmos dentro do módulo. Vejamos um exemplo:

```
module Payment
  MIN_COST_FOR_FREE_SHIPPING = 10.00

  class Purchase
    attr_accessor :total_cost

    def calculate_shipping!
      if total_cost >= MIN_COST_FOR_FREE_SHIPPING
        puts "Parabéns, frete grátis para você!"
      else
        puts "Sem frete grátis :-("
      end
    end
  end
end

purchase = Payment::Purchase.new
purchase.total_cost = 15.00
purchase.calculate_shipping! # Parabéns,
                             # frete grátis para você!

puts MIN_COST_FOR_FREE_SHIPPING # uninitialized constant
puts Payment::MIN_COST_FOR_FREE_SHIPPING # 10.0
```

Em algumas situações é possível que duas classes de mesmo nome sejam acessíveis em um escopo. Para isso, é possível explicitar integral ou parcialmente o escopo usando o `::`. Caso o escopo seja o escopo raiz, é possível declarar da seguinte forma: `::File`. Isso irá fazer com que a resolução de nomes do Ruby vá buscar a classe ou módulo `File` no nível raiz, sem nenhum módulo.

Para entender melhor como organizar projetos e escopos, veja o Capítulo 8 do livro *Ruby Best Practices [3]*.

Módulos como mixins

Mixins é uma das funcionalidades mais importantes do Ruby. Além de classes, é possível declarar métodos de instância dentro de um módulo. Assim, é possível "misturar" esses métodos em qualquer classe ou objeto. Dessa forma, é possível fazer diversas classes obterem um conjunto de comportamento comum com apenas uma linha de código.

Dois exemplos básicos de *mixins* em Ruby são os módulos `Comparable` e `Enumerable`. O módulo `Comparable`, ao ser incluído em uma classe, basta que a classe implemente o método `<=>` (também conhecido como *spaceship method*) e ela ganha diversos outros comportamentos de graça, como `<`, `<=`, `==`, `>`, `>=` e `between?`. Com o `Enumerable`, implementar o método `each` faz com que a classe ganhe várias funcionalidades, como `all?`, `any?`, `map`, `count`, `detect` e outros. Veja a documentação do Ruby sobre `Enumerable` para mais detalhes (http://www.ruby-doc.org/core-1.9.3/Enumerable.html).

Contudo, vamos a um exemplo simples para entender como *mixins* funcionam:

```
module Shipping
  CUBED_WEIGHT_FACTOR = 167

  def dimensional_weight
    width * depth * height * CUBED_WEIGHT_FACTOR
  end
end

class ShippingPrice
  include Shipping

  attr_accessor :width, :depth, :height
end

shipping = ShippingPrice.new
shipping.width = 0.5;
shipping.depth = 0.8;
shipping.height = 0.3;

shipping.dimensional_weight # 20.04
```

Para criar *mixins*, basta declarar um módulo e construir métodos diretamente nele. Quando fazemos `include` de um módulo dentro de uma classe, seus métodos são "misturados", e portanto temos acesso como se o método estivesse implementado diretamente na classe. O mesmo acontece com os métodos do módulo. Como a verificação da existência de métodos é apenas em tempo de execução, este "contrato" de implementação não é verificado pelo interpretador, simplificando, portanto, a implementação.

Uma característica importante é que, quando fazemos um *mixin*, o módulo acaba se tornando um ancestral da classe que inclui este módulo. Isso nos dá uma série de

vantagens, tal como a verificação de tipos:

```
shipping.is_a? Shipping # true
```

O mesmo pode ser feito com módulos como `Enumerable`. Esta é uma forma de garantir que um objeto responde a uma série de métodos de maneira uniforme.

Mixins com extend

Existe uma outra forma de fazer *mixins*, com a palavra-chave `extend`. A diferença é que os métodos são incluídos a nível de classe, e não mais de instância:

```
module Builder
  def build(attributes={})
    new_object = new
    attributes.each do |name, value|
      # O código abaixo é o mesmo que
      # new_object.name = value
      new_object.send "#{name}=", value
    end

    new_object
  end
end

class ShippingPrice
  extend Builder
  attr_accessor :width, :height, :depth
end

shipping = ShippingPrice.build({
  :width => 0.8,
  :height => 0.2,
  :depth => 0.3
})

shipping.width # 0.8
```

O código anterior usa um pouco do que chamamos em Ruby de *metaprogramação*, ou seja, programação para gerar código. O módulo `Builder` basicamente cria um construtor em que um `Hash` é convertido para atributos de um objeto

via métodos de escrita (métodos terminados com =). Usamos então o `extend` para misturar a funcionalidade de construção ao nível da classe.

> **Meta-programação**
>
> Meta-programação é uma funcionalidade importante para quem escreve bibliotecas e sistemas avançados. Como este é um livro para quem está começando, meta-programação pode dar um nó na cabeça e portanto não vamos ver muito mais detalhes sobre o assunto.
>
> Porém, se você se sente um programador aventureiro, recomendo a leitura do *Metaprogramming Ruby [8]*, um livro excelente para quem quer iniciar nas artes às vezes misteriosas da meta-programação.

Exceções

O controle de exceções em Ruby acontece em 3 fases. Primeiro, um bloco de código, iniciado pela cláusula `begin` é executado. Em seguida, caso alguma exceção aconteça (via erros de sintaxe Ruby ou pela cláusula `raise`), o interpretador Ruby vai buscar em todas as possíveis cláusulas `rescue` qual é pertinente para o tratamento daquela exceção. Isso é feito através da comparação de classes, ou sua hierarquia. E, por fim, se existir, o bloco `ensure` é executado, independente da ocorrência ou não de uma exceção.

Sim, muita coisa pode acontecer em um pedaço pequeno de código, então vamos com calma para entender cada passo. Primeiro, vejamos como tratar exceções genericamente.

```
def calculate_installment_price(total_value, installments)
  begin
    puts "O resultado é #{total_value / installments}"
  rescue
    puts "Não foi possível calcular o valor da parcela"
  end
end

calculate_installment_price(100, 5) # O resultado é 20.0
```

```
calculate_installment_price(100, 0) # Não foi possível calcular
                                    # o valor da parcela
```

Neste caso, quando passamos um valor inválido para o número de parcelas (*installments*), uma exceção é disparada, e a execução é interrompida, levando a execução do programa para o bloco associado ao `rescue`. Por essa razão, a impressão do texto *"O resultado é ..."* não é executada. É importante notar que nesse caso não importa qual exceção seja disparada, o bloco `rescue` será executado.

É possível, porém, associar um bloco `rescue` diretamente a uma classe. Dessa forma, dependendo do erro que ocorrer no bloco associado, podemos tratá-lo de maneira diferente. Vamos usar essa ideia para mostrar um erro diferente para o usuário:

```
def calculate_installment_price(total_value, installments)
  begin
    puts "O resultado é #{total_value / installments}"
  rescue ZeroDivisionError
    puts "Número de parcelas deve ser > 0"
  rescue
    puts "Não foi possível calcular o valor da parcela"
  end
end
calculate_installment_price(100, 5) # O resultado é 20.0

calculate_installment_price(100, 0) # Número de parcelas
                                    # deve ser > 0

calculate_installment_price("", 0)  # Não foi possível calcular
                                    # o valor da parcela
```

Dessa forma, quando enviamos o como número de parcelas, a exceção gerada pelo interpretador é a `ZeroDivisionError`, que é tratada pelo primeiro bloco `rescue`. No segundo caso, porém, este bloco não corresponde à exceção gerada (`NoMethodError`), portanto o último bloco é executado.

É bastante comum termos que garantir que algo aconteça mesmo que haja erro durante um processo, como liberar recursos. Por isso, temos o bloco `ensure`. O bloco `ensure` sempre será executado, independente da ocorrência de exceções. Inclusive, o bloco `ensure` pode ocorrer mesmo sem blocos `rescue`.

```
def calculate_installment_price(total_value, installments)
  begin
```

```
    puts "O resultado é #{total_value / installments}"
  rescue
    puts "Não foi possível calcular o valor da parcela"
  ensure
    puts "Pronto."
  end
end

calculate_installment_price(100, 5) # O resultado é 20
                                    # Pronto.

calculate_installment_price(100, 0) # Não foi possível calcular
                                    # o valor da parcela
                                    # Pronto.
```

O `begin` é bastante útil caso queiramos especificar um trecho de código que pode sinalizar exceções. Porém, no caso de métodos inteiros, podemos omiti-lo:

```
def calculate_installment_price(total_value, installments)
  puts "O resultado é #{total_value / installments}"
rescue
  puts "Não foi possível calcular o valor da parcela"
ensure
  puts "Pronto."
end
```

Esta definição é um pouco mais enxuta, sem perder clareza, então é recomendado o uso quando possível.

Mais informações sobre exceptions

Tratamento de exceções, por mais simples que pareça, é um assunto delicado, e é necessário bastante cuidado para que o código não se torne um grande macarrão incompreensível.

Se você quiser saber mais sobre o assunto, recomendo a leitura do excelente livro *Exceptional Ruby [6]*.

2.7 Bibliotecas e RubyGems

Uma linguagem de programação não pode ser completa sem possuir um mecanismo de compartilhamento de código, de modo que possamos quebrar um sistema grande em diversos arquivos.

Bibliotecas

Em Ruby, existem três maneiras para se carregar um código externo:

- `load`: Carrega o arquivo referenciado;
- `require`: Carrega o arquivo referenciado, porém, se ele já foi carregado em algum outro ponto do código, nada será feito;
- `require_relative`: Funciona da mesma maneira que o `require`, porém somente funciona com caminhos relativos ao sistema de arquivos. Veremos mais detalhes logo adiante.

O `require` é definitivamente a maneira mais usada para carregar arquivos Ruby pelo fato do arquivo ser carregado apenas uma vez. Quando um arquivo é carregado via `require`, ele é adicionado a uma lista chamada `$LOADED_FEATURES`. Ou seja, isso significa que se o arquivo Ruby for editado externamente, ele não será recarregado, havendo a necessidade de reiniciar o processo.

Uma particularidade sobre o `require` é que, se o caminho passado for relativo, ele será procurado apenas no `$LOAD_PATH` e um erro será apresentado caso não seja encontrado. Para carregar arquivos que não estão no `$LOAD_PATH`, é necessário mencionar o caminho relativo. Vejamos a seguir exemplos:

Usando o `load`:

```
load 'purchase.rb' #=> true
Purchase.new       # #<Purchase:0x007f9062b49b60>
```

Agora o `require`:

```
require './purchase' # Podemos omitir o .rb com o require!
Purchase.new       # #<Purchase:0x007f9062b49b60>
```

Note que se removermos o caminho do arquivo, o `require` necessita que a pasta em que o arquivo se encontra esteja no `$LOAD_PATH`:

```ruby
require 'purchase' # LoadError: cannot load such file -- purchase
$LOAD_PATH << "./" # $LOAD_PATH é um Array com todos os caminhos
                   # a serem procurados

require 'purchase' # Agora funciona, o purchase.rb é encontrado
Purchase.new       # #<Purchase:0x007f9062b49b60>

require_relative 'purchase'
Purchase.new                # #<Purchase:0x007f9062b49b60>
```

Cuidado com require_relative

O `require_relative` possui um pequeno problema quando executamos o comando via IRB. Isso acontece porque, quando executamos o `require_relative`, o Ruby tenta inferir a pasta raiz pelo arquivo em execução. O problema é que, quando executamos via IRB, não existe nenhum arquivo, então o Ruby irá sinalizar uma exceção mencionando este fato.

RubyGems

A linguagem Ruby possui uma comunidade bastante ativa de programadores que compartilham código. Para facilitar esse compartilhamento, a comunidade Ruby inventou as chamadas RubyGems, ou "gems". A criação de uma RubyGem ocorre da seguinte forma:

1) Um desenvolvedor constrói uma biblioteca e deseja compartilhá-la;

2) O desenvolvedor prepara um arquivo chamado `gemspec` que informa coisas como nome do autor original, colaboradores, versão daquela biblioteca, site, documentação, e várias outras informações;

3) Em seguida, ele prepara o pacote usando a ferramenta `gem`, que gera um arquivo terminado em `.gem`.

4) Depois de criar uma conta no site do RubyGems (http://rubygems.org), gratuitamente, o desenvolvedor usa o comando `gem` para publicá-la.

Instalando Rubygems

Uma vez publicada, qualquer desenvolvedor pode instalar e usar essa gem. Vamos instalar a gem `nokogiri`, para processamento de XMLs. Para isso, primeiramente temos que instalá-la, usando a linha de comando:

```
$ gem install nokogiri
Fetching: nokogiri-1.5.3.gem (100%)
Building native extensions.  This could take a while...
Successfully installed nokogiri-1.5.3
1 gem installed
Installing ri documentation for nokogiri-1.5.3...
Installing RDoc documentation for nokogiri-1.5.3...
```

No caso do `nokogiri`, é necessário compilar código C associado à RubyGem, por isso o comando `gem` exibe "Building native extensions ...". Depois de instalado, o RubyGems gera documentação via `ri` e `RDoc`. Não entraremos em detalhes como esses aplicativos funcionam pois usaremos documentações online que facilitam a busca de informações.

Depois de instalado, é possível usar o `nokogiri` da seguinte forma:

```
require 'rubygems'
require 'nokogiri'

doc = Nokogiri::HTML('<html></html>')
```

> **REQUIRE 'RUBYGEMS' RETORNA FALSO**
>
> Não fique preocupado se o `require 'rubygems'` retornar falso. Como Rubygems é muito comum, muitas ferramentas podem fazer o require para você. O que importa mesmo é que o require da gem que você está usando é feito com sucesso.

É importante garantir que façamos o `require` do RubyGems para que ele possa configurar o ambiente Ruby de modo a encontrar as bibliotecas instaladas, mas talvez isso não seja suficiente. Para instalar e usar uma gem, é recomendado que você leia a documentação disponibilizada na página da gem, pois nem sempre a instalação é exatamente igual à que vimos anteriormente.

Escolhendo uma RubyGem

A comunidade Ruby é bastante efusiva em bibliotecas, então é muito provável que já exista uma gem para resolver o seu problema! Mas antes de sair usando uma gem que você encontrou, verifique os seguintes pré-requisitos:

- Existe uma comunidade ativa de desenvolvedores, ou seja, há atividade recente de alterações ou não há uma lista grande de bugs no tracker do projeto;
- Verifique se a biblioteca possui testes unitários e melhor ainda, se a última compilação do projeto está OK (projetos populares executam suas compilações no Travis-CI [http://travis-ci.org/])

Outra dica é procurar por soluções no site RubyToolBox (https://www.ruby-toolbox.com/). Uma solução famosa, acompanhada e usada por muita gente é uma forma de tentar diminuir as chances de você encontrar bugs, pois alguém já deve ter se encontrado na mesma situação e resolvido o problema.

2.8 Fim!

Ufa! Você está lendo até aqui ainda? Que bom. Cobrimos bastante coisa de Ruby, mas é claro que a linguagem tem muito mais para oferecer. Durante o capítulo, fiz referência a diversos livros que podem trazer mais conhecimento em diversos aspectos de Ruby. É extremamente recomendado que você os leia depois que terminar de ler este livro, caso queira (espero que sim!) continuar sua carreira como desenvolvedor Ruby e Ruby on Rails.

Ruby é uma linguagem fascinante, e espero sinceramente que tenha gostado. Agora, vamos aprender o *framework* Ruby on Rails, e começar a fazer o Colcho.net do início ao fim.

Parte II

Ruby on Rails

Capítulo 3

Conhecendo a aplicação

Para cada fato existe um conjunto infinito de hipóteses. Quanto mais você observa, mais você enxerga.
– Robert Pirsig, em Zen and the Art of Motorcycle Maintenance

Então você já aprendeu bastante da linguagem Ruby para dar os primeiros passos, mas ainda temos muito o que aprender pela frente. Durante os próximos capítulos, vamos aprender mais da linguagem enquanto aprendemos o *framework* Ruby on Rails.

Mas antes de começar, você sabe qual a diferença entre um *framework* e uma biblioteca? Quando usamos uma biblioteca, nós programadores escrevemos nosso código, chamando a biblioteca quando necessário. A camada entre a biblioteca e o nosso código é bastante distinta.

O mesmo não acontece com um *framework*. Para se ter uma ideia, muitas vezes o ponto inicial do sistema não é um código que você escreve. Nosso código faz o intermédio com diversas outras bibliotecas que compõem o *framework*, de forma que o resultado torna-se mais poderoso do que a soma das partes.

O Ruby on Rails não é diferente. Ele é um *framework* usando uma estrutura chamada MVC -- *Model View Controller*, bastante conveniente para a construção de aplicativos Web. O Rails também é usado por muitos desenvolvedores já há bastante tempo, portanto já foi testado em diversas situações, como alta carga, ou grande número de usuários. É fácil começar com Rails pois ele faz muito trabalho por você, mas se aprofundar no Rails te dá ainda mais poder.

É muita coisa para aprender, mas não se preocupe, este livro está em seu poder justamente para facilitar esta tarefa. Vamos então ver como vamos modelar nossa aplicação e em seguida vamos ver como o Ruby on Rails vai nos ajudar a construí-la.

3.1 Arquitetura de aplicações web

Desenhar aplicações bem feitas em Ruby on Rails é um pouco mais complicado do que simplesmente criar páginas atrás de páginas. A razão disso é que ele é preparado para criar aplicações modernas e arrojadas. Isso significa que não somente deve responder HTML aos usuários, mas também responder de maneira adequada para aplicações ricas em *client-side*, interagindo com *frameworks* como Backbone.js (http://backbonejs.org/), Ember.js (http://emberjs.com/) ou outros.

O que são aplicações ricas em client-side?

Depois da repopularizarão do JavaScript com o uso intensivo de AJAX nas páginas Web, os browsers tem se tornado muito mais poderosos do que antigamente, inclusive com os benefícios incorporados pelo HTML 5. Tendo isso em mente, programadores estão criando aplicações cada vez mais complexas no browser.

Nos últimos anos, diversos *frameworks* tem sido lançados, tal como os mencionados anteriormente. Dessa forma, existem aplicações Web que a parte de *back-end* tornou-se uma API que está totalmente independente da apresentação final ao usuário.

3.2 Recursos ao invés de páginas

Para que nossas aplicações fiquem elegantes, precisamos parar de pensar na maneira antiquada de se criar aplicações Web cheias de páginas e interações complexas e pen-

sarmos em recursos. Não é uma mudança fácil, mas vamos usar este pensamento durante o livro todo, então até o final, você ficará mais confortável com essa ideia.

Se você já ouviu falar ou sabe o que é REST (*Representational State Transfer*, ou Transferência de estado de representação), entender como vamos usar recursos para modelar nossa aplicação será bem mais fácil. Se você quiser se aprofundar no assunto, recomendo a leitura do livro *Rest in Practice [7]*.

3.3 Recursos no Colcho.net

Então vamos lá, como fica um recurso no Colcho.net? Relembremos que o Colcho.net é um aplicativo em que os seus usuários podem mostrar quartos vagos, ou até um colchonete que pode ser usado em sua sala em troca de uma grana e quem sabe fazer novas amizades.

Assim, o primeiro recurso que podemos identificar é o "Quarto". Em nosso sistema, será possível listar quartos, exibir detalhes de um quarto, criar um novo quarto, editar um quarto já existente e removê-lo.

Recursos assim são fáceis de serem mapeados, pois estão diretamente ligados a uma unidade lógica do sistema. Porém, existem recursos que são menos óbvios. Por exemplo, como representar o login de um usuário? Para isso, teremos o recurso "Sessão". Poderemos criar uma sessão ou destruí-la, porém nada mais que isso.

Em sistemas REST, ações com recursos são mapeados em duas partes: um verbo HTTP e uma URL. Veja os seguintes exemplos:

O que são verbos HTTP?

Verbos HTTP são ações que podem ser executadas em um recurso identificado por uma URL. A implementação do comportamento fica a critério do servidor, não existe um padrão. Os verbos usados no Ruby on Rails são:

- **GET** - para retornar um recurso específico ou uma coleção;
- **POST** - para enviar um novo elemento à uma coleção;
- **PUT** - para alterar um elemento existente;
- **DELETE** - para remover um elemento.

- `GET /rooms`: Verbo `GET` na URL `/rooms` mapeia para a listagem de todos os quartos;
- `POST /rooms`: Verbo `POST` na URL `/rooms` mapeia para a inserção de mais um novo quarto na coleção de quartos;
- `DELETE /rooms/123`: Verbo `DELETE` na URL `/rooms/123` mapeia para a remoção do quarto cujo identificador único é "123".

Se fizermos dessa maneira, o Rails nos ajuda e bastante e de quebra ganhamos facilidade ao usar *frameworks* de *client-side* quando necessário.

Essa é a visão externa da arquitetura de aplicações Web. Vamos entrar então na arquitetura interna.

3.4 Conhecendo os componentes

O Ruby on Rails é um *framework* que adota o padrão de arquitetura chamado `Model-View-Controller (MVC)`, ou Modelo, Apresentação e Controle.

Modelos possuem duas responsabilidades: eles são os dados que normalmente ficam persistidos em um ou mais banco de dados (seu perfil de usuário, por exemplo). Eles também fazem parte da regra de negócio, ou seja, cálculos e outros procedimentos, como verificar se uma senha é válida.

O **Controle** é a camada intermediária entre a Web e o seu sistema. Ele pega os dados que vem de parâmetros na URL e/ou de um formulário e repassa para os modelos, que vão fazer o trabalho pesado. Em seguida, pega o resultado e transforma da maneira adequada para a Apresentação.

A **Apresentação** é como o aplicativo mostra o resultado das operações e os dados. Normalmente podem ser uma bela página usando as novas tecnologias de CSS 3 e HTML 5 a até pequenas representações de objetos em JSON.

3.5 Os modelos

Quando criamos aplicações Ruby on Rails, basicamente usamos um componente chamado `ActiveRecord`. O `ActiveRecord` é um *Object-Relational-Mapping*, ou seja, uma biblioteca que faz o mapeamento de estruturas relacionais (leia-se SQL e bancos de dados relacionais) a objetos Ruby.

O `ActiveRecord` internamente usa uma biblioteca bastante poderosa chamada `Arel`. Com o uso do `Arel` o `ActiveRecord` consegue transformar chamadas de métodos Ruby em complexas consultas SQL e depois mapear de volta em objetos Ruby. Isso tudo é muito conveniente, veja alguns exemplos:

```
Room.all
#  SELECT "rooms".* FROM "rooms"
Room.where(:location => ['São Paulo', 'Rio de Janeiro'])
# SELECT "rooms".* FROM "rooms" WHERE "rooms"."location"
#                               IN ('São Paulo', 'Rio de Janeiro')
```

Além disso, modelos `ActiveRecord` possuem outra característica importante. Eles possuem ferramentas para validação de atributos, *callbacks* em momentos oportunos (como antes de atualização ou criação), tradução e outros. Este componente é o `ActiveModel`.

Mas os modelos não necessariamente precisam ser objetos `ActiveRecord`. É bastante comum separarmos regras complexas em diversas classes em Ruby puro. Isso é importante para evitar que os modelos ou controles fiquem grandes e complexos. Vamos ver um exemplo disso no capítulo "10 Login do usuário".

No caso do Colcho.net, o Quarto se encaixa perfeitamente como um modelo. Assim, podemos criar validações como a presença de uma localidade ou calcular a disponibilidade dele em uma certa data.

3.6 Controle

A camada de controle é o intermédio entre os dados que vem dos usuários do site e os Modelos. Outro principal papel da camada de controle é gerenciar sessão e cookies de usuário, de forma que um usuário não precise enviar suas credenciais a todo momento que fizer uma requisição.

Após obter os dados na camada de modelos, é papel da camada de Controle determinar a melhor maneira de representar os dados, seja via a renderização de uma página HTML ou na composição de um objeto JSON.

No Ruby on Rails, os componentes que trabalham nessa camada são o `ActionDispatch` e o `ActionController`, ambos parte do pacote denominado `ActionPack`. O `ActionDispatch` trabalha no nível do protocolo HTTP, fazendo o *parsing* dos cabeçalhos, determinando tipos MIME, sessão, cookies e outras atividades.

Já o `ActionController` dá ao desenvolvedor o suporte para escrever o seu código de tratamento das requisições, invocando os modelos e as regras de negócio adequadas. Ele também dá suporte a filtros, para, por exemplo, verificar se um usuário está logado no sistema para fazer uma reserva no site.

Vamos trabalhar bastante com esses componentes no decorrer do livro e aprender como eles funcionam e como podemos implementá-los.

3.7 Apresentação

A camada de apresentação é onde prepara-se a resposta para o usuário, depois da execução das regras de negócio, consultas no banco de dados ou qualquer outra tarefa que sua aplicação deva realizar.

As maneiras mais comuns de se exibir dados atualmente na web é através de páginas HTML. No Ruby on Rails, conseguimos construí-las com o auxílio da biblioteca `ActionView` (também membro do `ActionPack`). Também é possível ter a representação JSON dos dados, via a serialização de objetos (transformação de objetos Ruby puro em JSON) ou via a construção de templates com o `JBuilder`, por exemplo.

Depois que a apresentação final é montada, ela é entregue ao usuário via um servidor web, como por exemplo, uma bela página mostrando todas as informações de um quarto, ou um objeto JSON que vai ser lido pelo JavaScript de uma outra página.

Por fim, o Rails ainda possui uma estrutura complexa para gerenciar imagens,

stylesheets e javascripts, chamada `Asset Pipeline`, pré-processando os arquivos e preparando-os para entregar da melhor forma para o usuário.

3.8 Rotas

Como o Ruby on Rails sabe qual Controle deve ser invocada para as requisições? Para resolver essa questão, é preciso indicar que uma determinada URL, quando invocada, dispara um Controle. O papel desse mapeamento é das Rotas, que são voltadas para recursos, como vimos no início deste capítulo. Mas é possível fazer outras diversas maneiras de rotas com a complexidade que desejar.

Veja a figura 3.1. Nela é possível ver a interação entre as diversas camadas MVC e o roteador quando o usuário requisita uma página de edição do recurso "quarto":

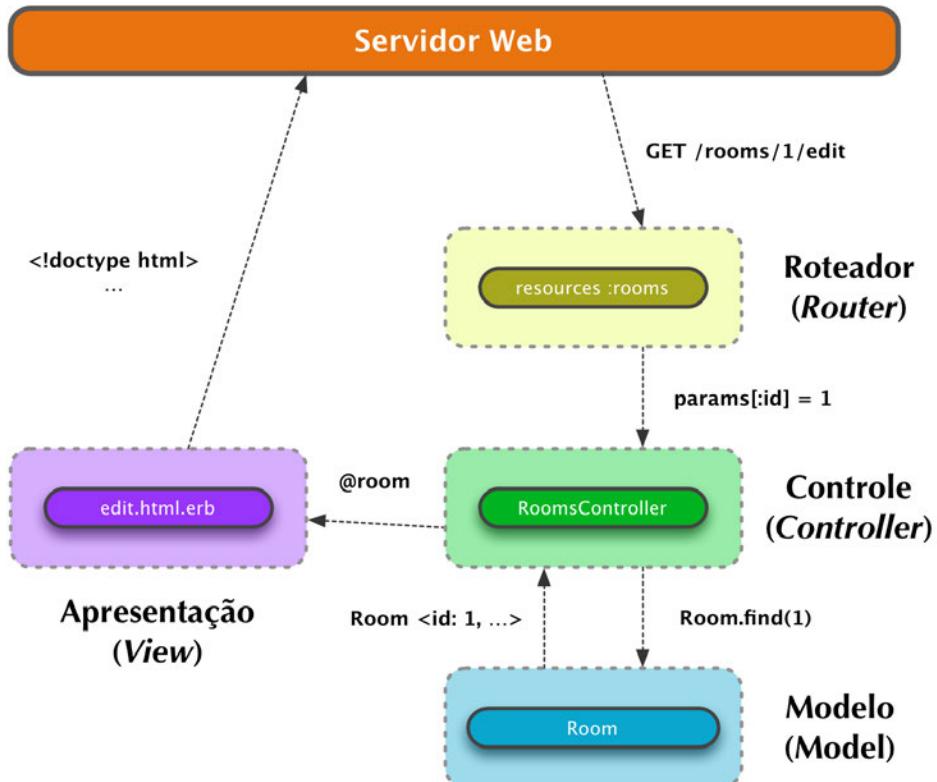



Figura 3.1: Fluxo do MVC

3.9 Suporte

Em volta de todo os componentes que mencionamos antes, existe também `ActionSupport`. É nele que ficam diversas ferramentas e extensões ao Ruby, para deixá-lo ainda mais fácil de usar. Graças a ele é que podemos fazer a seguinte linha de comando:

```
2.days.ago
# => Sun, 03 Jun 2012 01:08:33 BRT -03:00
```

Essa linha de código imprime a data de dois dias atrás.

Existem várias funcionalidades dentro do `ActionSupport`, mas não vale a pena entrar em detalhe. Vamos usá-lo diversas e você acabará aprendendo esses truques e recursos durante o livro.

3.10 Considerações finais

Não se preocupe em decorar todos os nomes que foram vistos nesse capítulo. Você pode consultar o livro sempre que precisar saber. É importante porém saber que eles existem para que você possa procurar ajuda ou documentação.

Aliás, falando em documentação, você aprenderá com o decorrer do livro que a API do Rails é bem extensa e cheia de métodos. Ambiente-se a desenvolver com o site da documentação oficial do Rails aberta. Ela está disponível em http://api.rubyonrails.org/. Sempre que tiver alguma dúvida sobre algum método do Rails, você pode procurar que estará lá.

O que é extremamente importante entender desse capítulo, porém, são as ideias de recursos e entender o que cada parte do MVC faz devidamente. Isso significa que, sempre que possível, devemos modelar nossas aplicações como recursos e que **nunca**, **jamais**, devemos colocar regras importantes de negócio em Controles, por exemplo, pois não é o papel dele.

Entendido isso, vamos continuar. Agora é hora de voltar a programar!

Capítulo 4

Primeiros passos com Rails

Conhecer seus limites já é estar a frente deles.
– Georg Wilhelm Friedrich Hegel

Para criar uma aplicação Rails, usaremos um *script* que a gem do Ruby on Rails instala no sistema. Se você ainda não instalou Ruby e Rails, dê uma olhada na seção 2.1, lá você encontrará instruções de como fazê-lo e em seguida volte aqui para começarmos a construção do Colcho.net.

4.1 Gerar o alicerce da aplicação

Vamos ao terminal gerar o esqueleto para o Colcho.net. Para criarmos um novo projeto, precisamos executar no terminal o comando `rails new`, passando em seguida o nome do projeto que será criado, no caso `colchonet`.

Ao começar a instalar, aproveite para pegar mais café ou sua bebida favorita, pois pode demorar um pouco.

```
$ rails new colchonet
      create
      create  README.rdoc
      create  Rakefile
      create  config.ru
      create  .gitignore
      ...
      Installing sass (3.1.19)
      Installing sass-rails (3.2.5)
      Installing sqlite3 (1.3.6) with native extensions
      Installing uglifier (1.2.4)
      Your bundle is complete! Use 'bundle show [gemname]'
               to see where a bundled gem is installed.
```

Como vimos no Capítulo 3, vamos escrever código **dentro** de templates e arquivos gerados pelo próprio Rails. A estrutura de pastas gerada quando criamos o projeto é bastante importante, e se parece com as figuras 4.1 e 4.2.

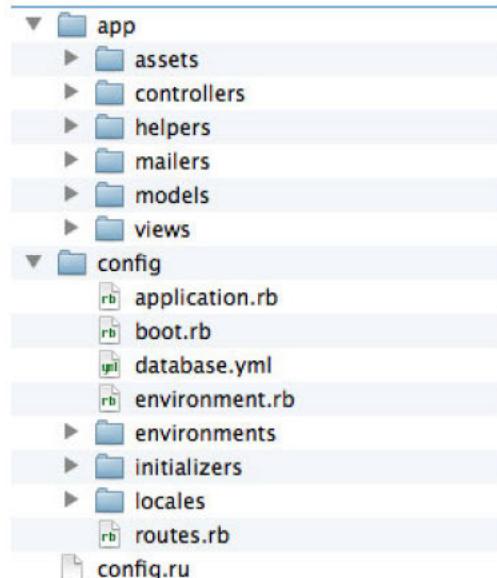



Figura 4.1: Estrutura de pastas geradas - 1

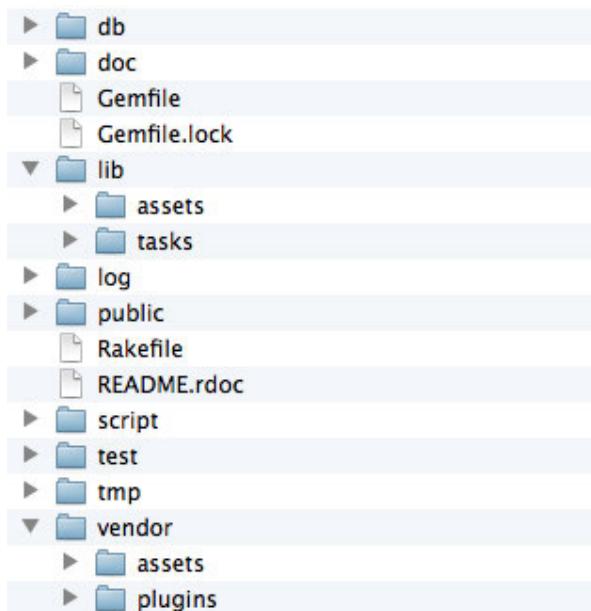



Figura 4.2: Estrutura de pastas geradas - 2

Cada pasta possui um significado e já é um efeito do conceito que vimos no capítulo 1, chamado "Convenção sobre Configuração". Isso significa que não teremos um XML que dirá em que pasta fica o quê, em contrapartida, significa também que temos que seguir a estrutura proposta. Eu pessoalmente acho que é uma ótima troca!

Vamos entender o que é cada uma delas:

- **app** - É onde fica o código principal da sua aplicação. Controles ficam na pasta *controllers*, apresentações em *views*, javascript e CSS em *assets* e modelos em *models*. A pasta *helpers* guarda códigos auxiliares usados em apresentações e *mailers* são classes para o envio de emails.

- **config** - Configuração da aplicação, como informações sobre banco(s) de dados, internacionalização de strings (I18n), time zone, e outras coisas.

- **db** - Armazena migrações e o esquema do banco de dados usado na aplicação.

- **doc** - Toda a documentação de sua aplicação deve estar aí.

- **lib** - Armazena código externo à sua aplicação mas que não faz parte de gems.

Aí também ficam tarefas `rake`, código que deve ser executado fora do ambiente Web.

- **log** - Onde ficarão seus logs e, se tudo der certo, uma pasta que você raramente terá que acessar.
- **public** - Arquivos estáticos como `favicon.ico`, `robots.txt`, `404.html` e `500.html`, que serão exibidos nos casos de página não encontrada e erro de servidor, respectivamente.
- **script** - Guarda o script `rails`, usado para gerar código (vamos ver mais detalhes sobre esse comando a seguir).
- **test** - Testes unitários, integração, funcionais e performance ficam aqui. Este é um tópico bastante complicado e merece seu próprio livro, então, ao invés de vermos o assunto pela metade, indico a leitura do *RSpec Book [5]* ou o *Guia Rápido de RSpec [9]*.
- **tmp** - Nesta pasta ficam arquivos temporários como PIDs, cache, entre outros.
- **vendor** - É onde você deverá instalar plugins de terceiros que não são gems. Também é uma boa ideia instalar eventuais plugins javascript e CSS aqui, para separar o código da sua aplicação.

Dê uma olhada no conteúdo das pastas e abra alguns arquivos, apenas por curiosidade, para quebrar o gelo, já que você e o Rails ainda são estranhos entre si. Dê uma atenção especial ao arquivo `config/application.rb`, o arquivo de configurações gerais da sua aplicação.

4.2 Os ambientes de execução

O Rails possui o conceito de ambientes de execução do aplicativo e cada um deles possui um conjunto próprio de configurações. Um exemplo de configuração que varia por ambiente é o *caching* de classes, ou seja, todo o código Ruby presente na pasta `app` será carregado apenas uma vez. Isso é bastante útil em produção devido a maior performance, porém, em desenvolvimento, atrapalha bastante.

Por padrão, o Rails possui, mas não se limita a, três ambientes:

- `development`: É o ambiente padrão, onde o Rails é totalmente configurado para facilitar o desenvolvimento;

- `test`: Ambiente para se executar testes de todos os níveis (unitários, integração, etc.);
- `production`: É o ambiente em que a aplicação deve executar quando rodando no servidor, para os seus usuários. Por esse motivo, é otimizado para performance.

Cada ambiente possui seu próprio banco de dados. O arquivo `config/database.yml` possui a definição da configuração de banco de dados para cada ambiente. Veja o exemplo para o ambiente de desenvolvimento, usando o adaptador para o PostgreSQL:

```
development:
  adapter: postgresql
  database: colchonet_dev
  host: localhost
  username: vinibaggio
  password:
  pool: 5
  timeout: 5000
```

Outros arquivos relacionados com ambientes ficam na pasta `config/environments`, no qual cada arquivo é carregado automaticamente de acordo com o ambiente em execução.

Variável de ambiente RAILS_ENV

A variável de ambiente `RAILS_ENV` é usada para determinar em que ambiente o Rails irá executar, por padrão sendo `development`. Assim, é possível executar comandos específicos para cada ambiente, basta fazer, por exemplo:

```
RAILS_ENV=production rake db:create
```

4.3 Os primeiros comandos

Com seu terminal aberto na pasta do projeto (`colchonet`), execute o comando `rails server`:

Comandos no terminal

A partir de agora, vou mostrar para você diversos comandos de terminal e suas respostas. Por isso, nessas listagens você deve digitar apenas o que segue o símbolo $ (dinheiro), ou se por um acaso a linha for muito longa, será quebrada com o símbolo \(contra-barra). Exemplos:

```
$ rails server
=> Booting WEBrick
```

Nesse caso, você deverá digitar apenas `rails server` e apertar a tecla Enter. O que não tiver esse prefixo será uma saída aproximada do que você vai ver ao digitar o comando.

```
$ rails server
=> Booting WEBrick
=> Rails 3.2.8 application starting in development on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
[2012-06-08 23:38:43] INFO  WEBrick 1.3.1
[2012-06-08 23:38:43] INFO  ruby 1.9.3 (2012-04-20) [x86_64-darwin11.3.0]
[2012-06-08 23:38:43] INFO  WEBrick::HTTPServer#start: pid=77137
port=3000
```

Abra o seu browser em http://localhost:3000/ e *voilà*, você verá uma bela página preparada pela equipe do Ruby on Rails. Nessa página, há alguns passos, como usar o comando `rails generate` para criar modelos e controles e remover o arquivo `public/index.html`. Vamos seguir essa sugestão.

Figura 4.3: Seja bem vindo ao Ruby on Rails

O comando `rails generate` é um comando bastante útil. Se você digitá-lo sem parâmetros, pode ver uma lista do que pode ser gerado. Essa lista pode aumentar, dependendo das gems que você instalar no Ruby on Rails. Vamos ver isso no futuro.

O Rails inclui várias ferramentas para você e esse *script* de geração de templates é um ótimo exemplo de como a ferramenta te ajuda a automatizar tarefas repetitivas. Nesse espírito, vamos agora começar a criar nosso primeiro recurso no sistema, o recurso quarto, ou `Room`, que terá um título, uma descrição e uma localização. Faremos isso através do comando `rails generate scaffold`, que irá gerar vários arquivos para nós:

```
$ rails generate scaffold room title location description:text

      invoke  active_record
      create    db/migrate/20120609065934_create_rooms.rb
      create    app/models/room.rb
      invoke    test_unit
      create      test/unit/room_test.rb
```

```
create      test/fixtures/rooms.yml
invoke  resource_route
 route    resources :rooms
invoke  scaffold_controller
create    app/controllers/rooms_controller.rb
invoke    erb
create      app/views/rooms
create      app/views/rooms/index.html.erb
create      app/views/rooms/edit.html.erb
create      app/views/rooms/show.html.erb
create      app/views/rooms/new.html.erb
create      app/views/rooms/_form.html.erb
invoke    test_unit
create      test/functional/rooms_controller_test.rb
invoke    helper
create      app/helpers/rooms_helper.rb
invoke      test_unit
create        test/unit/helpers/rooms_helper_test.rb
invoke  assets
invoke    coffee
create      app/assets/javascripts/rooms.js.coffee
invoke    scss
create      app/assets/stylesheets/rooms.css.scss
invoke  scss
create    app/assets/stylesheets/scaffolds.css.scss
```

Uou! O que aconteceu? Olhe a quantidade de arquivos que o Rails criou em um simples comando. Se você está curioso para saber o que é cada arquivo, vamos com calma. Vejamos primeiro detalhadamente o que é o comando que executamos através da figura 4.4:

Figura 4.4: Parâmetros do comando rails

Aqui podemos ver como o comando foi composto. Fazemos o `generate` de um *scaffold*, ou andaime. *Scaffolds* são interessantes pois já geram desde o modelo até a apresentação e páginas HTML, para que possamos modificar para ter o resultado que quisermos.

Em seguida, especificamos o nome do recurso e seus atributos. Por padrão, todos os campos serão "string", ou um campo de texto curto a não ser que seja especificado, como é o caso de `description`, a descrição de um quarto pode ser bem longa. Para isso, podemos usar o `:` para especificar o tipo daquela informação.

O comando *scaffold* já gera tudo que é necessário para que possamos fazer as operações mais comuns em um recurso, o famoso CRUD. CRUD nada mais é do que *create*, *retrieve*, *update* e *delete*, ou criar, buscar/listar, editar e remover. Para tal, primeiro precisamos criar o banco de dados. Não se preocupe em configurar um servidor MySQL ou PostgreSQL por enquanto, pois vamos usar um banco de dados bom e simples para o nosso propósito no momento, o SQLite. Para criar o banco de dados e deixar tudo pronto para ver o resultado do *scaffold*, basta executar, na pasta do projeto, os comandos `rake db:create` e `rake db:migrate`:

```
$ rake db:create

$ rake db:migrate
==  CreateRooms: migrating =================================
-- create_table(:rooms)
   -> 0.0019s
==  CreateRooms: migrated (0.0020s) ========================
```

O primeiro irá criar o banco de dados, se não existir. No caso do SQLite, o comando irá criar os arquivos `db/development.sqlite3`, `db/test.sqlite3` e `db/production.sqlite3`.

Em seguida, o `rake db:migrate` irá criar a tabela `rooms` no banco de dados usando a migração gerada pelo gerador. Veremos no futuro o que é uma migração, não se preocupe se você não souber o que é isso.

Comando rake

Observe que os dois últimos comandos são diferentes dos comandos que executamos neste capítulo. O `rake`, ou "ruby make", é uma ferramenta para executar tarefas de manutenção de uma biblioteca ou aplicação, independente do Rails. O Rails, porém, adiciona diversas tarefas pertinentes, como criação e migração de banco de dados, executar testes unitários, entre outros. Para saber todos os comandos `rake` que você pode executar no seu aplicativo, execute `rake -T`.

O comando `rails`, porém, em geral executa *scripts* que não dependem do ambiente da sua aplicação, ou seja, não dependem de conexão com banco de dados ou carregar o seu código Ruby, com exceção do `server`.

Agora, vamos executar o servidor:

```
rails server
```

Abra o seu browser favorito em http://localhost:3000/rooms e você será apresentado pela tela padrão gerada pelo *scaffold*:

Listing rooms

Title Location Description

New Room

Figura 4.5: Página de quartos

Aproveite para dar uma navegada, clique nos links e preencha os formulários. Legal né? Bastante funcional, apesar de simples. Agora vamos passo a passo entender o que aconteceu no gerador do *scaffold*.

Logs

Toda a requisição que você fizer, você vai ver que o servidor do Rails mostrar um **monte** de coisas, desde consultas SQL executadas até quando o servidor serve arquivos estáticos. Todo este texto será enviado também para a pasta `log`, no arquivo com o nome do ambiente em execução (por exemplo, `log/development.log`):

```
Started GET "/rooms" for 127.0.0.1 at 2012-08-14 21:58:15 -0700
Processing by RoomsController#index as HTML
  Room Load (0.2ms)  SELECT "rooms".* FROM "rooms"
  Rendered rooms/index.html.erb within layouts/application (2.1ms)
Completed 200 OK in 12ms (Views: 10.3ms | ActiveRecord: 0.2ms)
```

É sempre importante estar de olho nessas requisições. Podemos observar os parâmetros enviados pelo usuário, as consultas SQLs resultantes, o tempo gasto em cada parte da aplicação e o código de resposta enviado ao usuário.

4.4 Os arquivos gerados pelo scaffold

Vimos há pouco tempo que o `scaffold` gerou vários arquivos para nós. Mas o que significa cada um deles? O que eles fazem e qual seu impacto no nosso projeto?

Migrações

Migrações são pequenos *deltas* de um banco de dados, ou seja, elas registram as modificações que o *schema*, ou esquema, do banco de dados vai sofrendo. Elas contém três informações importantes: uma informação de versão, um código de regressão e um código de incremento e todas elas ficam dentro do diretório `db/migrate` do projeto.

Uma migração então possui um método `up`, que será executado quando o banco for migrado de uma versão à outra. Quando houver a necessidade de se regredir uma versão, talvez por causa de um bug, o método executado deverá ser o `down`. O nome da versão da migração encontra-se no nome do arquivo, que nada mais é do que uma data (ano, mês, dia, hora, minuto e segundo).

O Rails já conhece também o equivalente de desfazer para alguns comandos, como `drop_table` para desfazer um `create_table`. Assim, é possível criar o método `change`, que substitui ambos `up` e `down`. Você escreve o comando para incremento de versão e o Rails encontra o equivalente quando a versão for decrementada, poupando retrabalho pelo desenvolvedor.

Veja a pasta `db/migrate`. Lá haverá um arquivo que inicia com uma data, que depende da hora que você executou o comando, e termina com `_create_rooms.rb`:

```
class CreateRooms < ActiveRecord::Migration
  def change
    create_table :rooms do |t|
      t.string :title
      t.string :location
      t.text :description

      t.timestamps
    end
  end
end
```

Vimos que, para executar as migrações pendentes, basta executar:

```
rake db:migrate
```

No capítulo 5, vamos revisitar e aprofundar em outros comandos de migrações.

Modelo

O arquivo `room.rb` gerado pelo *scaffold* no diretório `app/models` é o modelo que tem seus dados guardados no banco de dados quando você clica no botão "Create Room". Nesse arquivo também ficam as regras de validação, relacionamentos com outros modelos e outras coisas. O código atual é simples e veremos como customizá-lo no futuro:

```
class Room < ActiveRecord::Base
  attr_accessible :description, :location, :title
end
```

Os próximos dois arquivos são relacionados a testes.

Teste unitário

Testes unitários são artefatos muito importantes para um sistema de qualidade. Escrever teste, porém, é algo que é bastante complicado para iniciantes, pois existe muita subjetividade em como escrever um teste. Por exemplo, devemos escrever um teste antes do código de produção (ou seja, código que vai ser de fato executado quando o site estiver no ar) ou depois? Devo usar qual técnica? Como usar `mocks` e quando usar `stubs`?

Esse assunto é bastante complicado e importante, portanto recomendo a leitura dos livros *The RSpec Book, [5]* e o *Test-Driven Development: By Example [2]*. A comunidade Rails leva testes tão a sério que o próprio *framework* vem com um conjunto de ferramentas para auxiliar o desenvolvedor a criar criá-los.

Um exemplo disso são os arquivos gerados. Toda vez que o Rails gera um modelo, controle ou apresentação, arquivos equivalentes para cada unidade são criados. Outro artefato gerado são as *fixtures*, ou acessório, em uma tradução grosseira. As *fixtures* são arquivos no formato YAML (http://yaml.org/), uma linguagem simples para serialização de dados, que será inserido no banco de dados antes de executar os testes unitários. Eles são úteis para podermos testar consultas ou para construir um cenário em que um código deve ser executado para que tudo funcione (por exemplo, para testar uma autenticação, é necessário que haja um usuário no banco de dados).

Rotas

Prosseguindo, a próxima modificação é no arquivo de rotas. O arquivo de rotas é onde o Rails mapeia a URL da requisição à um controle que você escreve. Se você se lembra bem, falamos na seção 3.2 que o Rails é bastante voltado à recursos, e agora vemos bem isso olhando o arquivo `routes.rb` no diretório `config`:

```
resources :rooms
```

Essa linha é tudo o que você precisa dizer para ganhar as seguintes rotas:

- `GET /rooms` - ação **index** - lista todos os quartos disponíveis;
- `GET /rooms/new` - ação **new** - mostra uma página com um formulário para a criação de novos quartos;
- `POST /rooms` - ação **create** - cria um novo recurso na coleção de quartos;

- `GET /rooms/:id` - ação **show** - exibe detalhes de um quarto cuja chave primária (famoso id) seja especificada na URL (id 123 para a URL /rooms/123, por exemplo)

- `GET /rooms/:id/edit` - ação **edit** - exibe o formulário para a edição de um quarto;

- `PUT /rooms/:id` - ação **update** - altera alguma informação do recurso cujo id seja especificado pelo parâmetro :id;

- `DELETE /rooms/:id` - ação **destroy** - destrói o objeto identificado pelo parâmetro :id

Essas ações são geradas por padrão e são todas as ações mais básicas que queiramos fazer em um recurso (o "CRUD"). É claro que dá para customizar, remover algumas dessas ações ou fazer URLs mais interessantes, mas sempre que fizermos recursos dessa forma, vamos poupar trabalho.

Controle

O próximo é o controle em si. Se você abrir o controle em um editor de código, vai ver exatamente as sete ações mencionadas anteriormente e um código básico que o Rails já gera para você. Vamos entrar em mais detalhes sobre controles no futuro.

Apresentações

Em seguida, temos as seguintes páginas, apresentações de *algumas* das ações mencionadas:

```
create      app/views/rooms/index.html.erb
create      app/views/rooms/edit.html.erb
create      app/views/rooms/show.html.erb
create      app/views/rooms/new.html.erb
create      app/views/rooms/_form.html.erb
```

Há três coisas importantes nessa lista. A primeira é que você pode ver que que os arquivos são todos terminados em "erb". O ERB é a linguagem de *templating* padrão do Rails, mas é possível usar outras. Vamos usar ERB durante esse livro, então não se preocupe com outra linguagem de template por enquanto. Ela não é nada de especial na verdade, é o Ruby embutido dentro de um arquivo HTML.

A segunda observação é que não há sete ações e que não necessariamente todas as páginas mapeiam para uma ação no controle/rota. Isso deve-se ao fato que algumas rotas não possuírem uma apresentação em si, apenas redirecionam ou apresentam outras. Um exemplo é a ação **create**, pois quando há sucesso, o usuário é redirecionado à ação **index**, e quando há erro, o formulário da ação **new** é reapresentado.

A terceira e última é que há um arquivo iniciado com `_`. Isso significa que essa página é uma *partial*, ou seja, o conteúdo dela é embutido em outras páginas. Nesse caso, o `_form.html.erb` é incluído tanto na página `new.html.erb` quanto em `edit.html.erb` pois o formulário é o mesmo, evitando problemas de `Ctrl-C Ctrl-V` e duplicidade de código. Alterando em um lugar, as mudanças se refletem em todas as páginas que usam essa *partial*.

Agora o projeto está criando vida. Vamos então à nossa primeira funcionalidade: cadastro de usuários.

Parte III

Mãos à massa

Nossa aplicação está começando a tomar alguma forma. Já temos um recurso, quartos, e já podemos cadastrá-lo, editá-lo e removê-lo e fizemos tudo isso com a ajuda do Rails.

Mas agora que já temos uma breve noção da estrutura do nosso projeto e também do próprio *framework*, podemos começar a construir uma funcionalidade do início ao fim, e bastante importante para o Colcho.net: cadastro e autenticação de usuários, o login.

Primeiramente, vamos construir o modelo de usuários. Vamos garantir integridade das informações, ou seja, construir validações para que um usuário cadastrado com sucesso no site possua todas as informações necessárias.

Uma vez que tivermos a base do modelo pronta, vamos montar as páginas de cadastro e edição de usuário e por fim visualização de perfil. Durante esse trabalho, vamos montar as rotas e construir as ações do controle sem o auxílio de geradores.

Mais para frente, vamos aprimorar o modelo, garantindo que o usuário possua uma senha encriptada no banco de dados, para que ninguém, nem mesmo os administradores do sistema, consigam acesso a uma informação tão sigilosa.

Os templates estarão bem crus, então vamos aplicar um pouco de CSS e HTML para tornar as telas mais interessantes. Vamos aprender como fazer tudo isso, e, em seguida, iremos aprender a usar o sistema de internacionalização do Rails para traduzir nossas páginas.

Para concluir o cadastro de usuários, vamos enviar um e-mail para o usuário para que ele possa confirmar sua conta, uma prática muito comum nos sites atuais para evitar cadastros falsos.

Capítulo 5

Implementação do modelo para o cadastro de usuários

É bom perseguir um objetivo para uma jornada, mas no fim o que realmente importa é a jornada em si.
– Ursula Guin

5.1 O usuário

Para o modelo de usuários, vamos precisar dos seguintes campos:

- Nome completo
- E-mail
- Senha
- Localidade

- Bio

Para gerar esse modelo, execute:

```
$ rails generate model user full_name email password location bio:text
```

> **Geradores**
>
> Você pode observar que muitas vezes vamos usar os geradores que o Rails provê. Porém, é completamente possível desenvolver sem o uso de nenhum gerador e criar todos os arquivos necessários na mão. Recomendo tomar esse caminho para quando você estiver mais experiente com o *framework* e tiver mais conforto para decidir o que é mais útil para você.

Vejamos a migração gerada:

```
class CreateUsers < ActiveRecord::Migration
  def change
    create_table :users do |t|
      t.string :full_name
      t.string :email
      t.string :password
      t.string :location
      t.text :bio

      t.timestamps
    end
  end
end
```

Era de se esperar ver SQL, já que estamos tratando de banco de dados, mas ao invés disso vemos Ruby. Isso é uma grande vantagem, já que infelizmente código SQL pode diferenciar entre sistemas de bancos de dados. O `ActiveRecord` já sabe lidar com essas diferenças e portanto apenas escrevemos em uma única linguagem comum.

A migração nada mais é do que uma classe que herda de `ActiveRecord::Migration`. Toda vez que o banco for migrado para uma versão superior, o Rails irá executar o que estiver no método `#change`, ou seja, criando a tabela `users` no banco de dados. Quando o banco for migrado para uma versão

inferior, o equivalente do método `create_table` será executado, o `drop_table`, removendo essa tabela.

Dentro do bloco associado ao método `create_table`, os campos serão criados com os tipos que declaramos no gerador. Por fim, o Rails adicionou por conta própria os campos relacionado à `timestamps`. São eles o `created_at` e o `updated_at`. O mais legal de tudo é que o `ActiveRecord` faz essa gestão para você automaticamente!

Altere o código da migração para que ela fique da seguinte forma:

```
class CreateUsers < ActiveRecord::Migration
  def change
    create_table :users do |t|
      t.string :full_name
      t.string :email
      t.string :password
      t.string :location
      t.text :bio

      t.timestamps
    end

    add_index :users, :email, :uniqueness => true
  end
end
```

Com essa alteração, estaremos criando um índice no campo `email` da tabela `users` com uma propriedade especial: unicidade. Vejamos o resultado disso em ação. Primeiro, faça:

```
$ rake db:migrate
==  CreateUsers: migrating =======================
-- create_table(:users)
   -> 0.0340s
-- add_index(:users, :email, {:uniqueness=>true})
   -> 0.0047s
==  CreateUsers: migrated (0.0341s) ==============
```

Isso irá criar a tabela no banco de dados e adicionar o índice. Agora vamos ao console do Rails, digitando `$ rails console`. Esse comando nos leva ao IRB, porém com todo o ambiente do Rails carregado, portanto é possível fazer alterações diretas ao banco de dados.

Atalhos para o comando rails

O comando `rails` possui alguns atalhos para não termos que digitar o nome completo do comando. Veja a lista de atalhos a seguir:

- `rails c` - equivalente ao `rails console`
- `rails s` - equivalente ao `rails server`
- `rails g` - equivalente ao `rails generate`
- `rails db` - equivalente ao `rails dbconsole` (abre o console do cliente de banco de dados, dependendo do qual estiver sendo usado)

```
User.create :email => 'admin@example.com'

# A saída é a seguinte (quebrada em linhas para facilitar a leitura):
#<User id: 1,
#       full_name: nil,
#       email: "admin@example.com",
#       password: nil,
#       location: nil,
#       bio: nil,
#       created_at: "2012-06-14 06:51:00",
#       updated_at: "2012-06-14 06:51:00">
```

Podemos observar algumas coisas importantes nesse pequeno exemplo. Primeiro é que o Rails, mesmo sem termos colocado nada na migração, criou o campo `id`. Essa chave primária é tão padrão no desenvolvimento de projetos que o Rails sempre vai criar para você nas migrações de criação de tabelas.

Podemos ver vários atributos como `nil`, o que é esperado, pois não os especificamos. Porém, podemos ver que o Rails automaticamente populou os campos `created_at` e `updated_at`. Esses campos também são automaticamente gerenciados pelo Rails, **caso** os campos existam, mas não é obrigatório.

Dica: console do Rails em modo sandbox

É muito comum fazermos alguns testes manualmente, criando, alterando ou deletando objetos do banco de dados. Para que nossos testes não fiquem poluindo o sistema, é possível iniciar o console usando uma chave para `sandbox`: `rails console --sandbox`. Quando você iniciar a sessão do console, o Rails irá emitir um início de transação no banco por todo o tempo. Quando você sair, um `ROLLBACK` será enviado ao banco, desfazendo todas as alterações.

Vamos dar uma espiada no modelo usuário, localizado em `app/models/user.rb`:

```
class User < ActiveRecord::Base
  attr_accessible :bio, :email, :full_name, :location, :password
end
```

A classe está praticamente vazia, com exceção de apenas uma linha de código. O `attr_accessible` é uma *class macro* que o `ActiveRecord` provê para adicionar um comportamento **muito** importante, porém peço a sua paciência, pois vamos entrar em detalhes sobre isso um pouco mais adiante, na seção 6.2.

Se você observar no exemplo que executamos no *console*, poderá ver que muitos campos ficaram em branco. Para evitar cadastros muito ruins ou até mesmo acidentes, vamos colocar algumas validações, garantindo que o usuário coloque todos os dados necessários.

5.2 Evite dados errados. Faça validações

O `ActiveRecord` nos disponibiliza diversos tipos de validações já prontas para grande parte das situações comuns, tais como validar presença, formato, inclusão em uma lista (masculino ou feminino, por exemplo), entre vários outros, além de disponibilizar ferramentas para criarmos as nossas próprias validações.

Tudo isso é feito de forma bastante conveniente, usando *class macros*. Vamos então validar:

- a presença do email, nome completo, localização e senha;

- a confirmação de senha, ou seja, o usuário precisa preencher um campo contendo a senha e outro para verificar se o usuário não cometeu erros de digitação;
- mínimo de 30 caracteres para a bio.

Vamos primeiro adicionar as validações de presença, as famosas informações obrigatórias:

```
class User < ActiveRecord::Base
  attr_accessible :bio, :email, :full_name, :location, :password

  validates_presence_of :email, :full_name, :location, :password
end
```

Salve o arquivo, pois vamos experimentar essas validações no console (basta executar novamente o comando `rails console`):

```
user = User.new
user.valid? # => false
user.save   # => false

user = User.new
user.email = 'joao@example.com'
user.full_name = 'João'
user.location = 'São Paulo, Brasil'
user.password = 'segredo'

user.valid? # => true
user.save   # => true
```

Quando executamos o método `#valid?` no modelo, o `ActiveRecord` irá executar a série de validações que o modelo tiver e retornar o resultado. Se o objeto estiver válido, retornará verdadeiro, e falso caso contrário, como é de se esperar.

O `#save` irá executar todas as validações (via o método `#valid?`) e se o modelo estiver válido, tentará salvar o modelo no banco de dados e retornar verdadeiro. Porém, se algum problema ocorrer no processo, seja validação ou salvar o objeto no banco de dados, método `#save` retornará falso.

De onde surgiram os métodos acessores?

Quando criamos nossos modelos e os fazemos herdar de `ActiveRecord::Base`, automaticamente, o Rails disponibiliza os métodos de leitura e escrita de todas as informações que a tabela possui.

Bom, já temos uma ideia de como funcionam as validações, então vamos adicionar uma nova:

```
class User < ActiveRecord::Base
  attr_accessible :bio, :email, :full_name, :location, :password

  validates_presence_of :email, :full_name, :location, :password
  validates_confirmation_of :password
end
```

Dica: recarregando as classes

Se você alterar o modelo e não reiniciar sua sessão do console do Rails, você não vai conseguir interagir com as suas novas alterações. Para não haver a necessidade de sair e entrar novamente no console, basta executar o comando `reload!` que o Rails irá recarregar tudo o que estiver dentro da pasta app.

O funcionamento do `validates_confirmation_of` é bem interessante. A partir do momento que adicionamos essa validação, para conseguirmos salvar um objeto no banco de dados, precisamos passar um novo atributo "virtual", ou seja, um atributo que não existe no banco de dados, chamado `password_confirmation`. Se ele não estiver igual ao campo de senha (`password`), o modelo não pode ser gravado no banco.

Com o mesmo console aberto, podemos executar:

```
reload!
# Reloading...
# => true
```

```
user = User.new
user.email = 'joao@example.com'
user.full_name = 'João'
user.location = 'São Paulo, Brasil'
user.password = 'segredo'
user.password_confirmation = 'errei_o_segredo'

user.valid? # => false
user.errors.messages
# => {:password=>["doesn't match confirmation"]}
```

Quando a validação é executada (tanto pelo método `#valid?` quanto pelo `#save`), o `ActiveRecord` popula um atributo especial no modelo, chamado `errors`. Com ele é possível verificar, em mensagens legíveis, quais foram os erros de validação.

Agora vamos a última validação, o tamanho da bio:

```
class User < ActiveRecord::Base
  attr_accessible :bio, :email, :full_name, :location, :password

  validates_presence_of :email, :full_name, :location, :password
  validates_confirmation_of :password
  validates_length_of :bio, :minimum => 30, :allow_blank => false
end
```

O `validates_length_of` faz diversos tipos de validação com tamanho de texto. Nesse caso, aplicamos ao atributo "bio" apenas duas restrições: o tamanho mínimo de 30 caracteres e ele não pode ser em branco. Porém, o `validates_length_of` aceita muitas outras opções, veja algumas delas:

- `:maximum`: Limita o tamanho máximo;
- `:in`: Ao invés de passar `:minimum` e `:maximum`, basta fazer, por exemplo, `:in => 5..10` para validar o mínimo de 5 e o máximo de 10 caracteres;
- `:is`: Limita o tamanho exato do texto;
- `:allow_blank`: Permite que o atributo fique em branco (ignorando as validações de tamanho);

Existem algumas outras opções menos usadas. Para saber quais são, recomendo olhar a documentação oficial do Rails.

O `ActiveRecord` ainda possui diversas outras validações que não usamos. Alguns exemplos, para você ter uma ideia:

- `validates_format_of`: Valida o formato de um texto com uma expressão regular;
- `validates_inclusion_of`: Valida a inclusão de um elemento em um enumerável, ou seja, um número em um *range* ou um texto dentre uma lista de opções;
- `validates_numericality_of`: Valida se o atributo passado é realmente um número, com algumas opções interessantes, por exemplo, `:less_than`, `:greater_than`, entre outros.

Além das validações citadas, ainda é possível criar validações customizadas.

Falando em validações customizadas, existe uma validação importante que devemos fazer mas ainda não fizemos. Sabe qual é? O e-mail. É possível colocar qualquer coisa, mesmo que não seja um e-mail válido.

Para isso, vamos usar uma validação simples de e-mail. Ela não é nem de longe ideal e não é compatível com o RFC de e-mails (veja em http://bit.ly/validacao-email uma expressão regular compatível, se estiver curioso), mas é suficiente para informar ao usuário que algo está errado. Já que vamos enviar um email de confirmação para o usuário poder usar o sistema, não é necessária tanta formalidade.

Para isso, vamos usar uma expressão regular extraída da biblioteca Devise (http://github.com/plataformatec/devise), uma biblioteca complexa de autorização de usuários. Apesar de completa, a biblioteca não é recomendada para quem está começando com Rails, portanto não vamos usá-la no Colcho.net.

Vamos também criar uma validação de unicidade para emails, ou seja, emails cadastrados não poderão existir previamente no site. É importante ressaltar que o `validates_uniqueness_of` possui um problema: primeiro o Rails verifica a existência do email a ser cadastrado e **depois** cria o modelo no banco, se o resto estiver OK. É possível que, entre a verificação e a criação, o email seja criado no banco e o Rails tentará criar o modelo de qualquer forma. É por isso que criamos a validação de unicidade também no banco de dados, e estamos criando essa validação no modelo apenas para *feedback* ao usuário.

Por fim, temos as últimas validações:

```
class User < ActiveRecord::Base
  attr_accessible :bio, :email, :full_name, :location, :password

  validates_presence_of :email, :full_name, :location, :password
  validates_confirmation_of :password
  validates_length_of :bio, :minimum => 30, :allow_blank => false
  validates_format_of :email, :with => /\A[^@]+@([^@\.]+\.)+[^@\.]+\z/
  validates_uniqueness_of :email
end
```

Sintaxe alternativa para validações

Existe uma outra sintaxe para validações. Veja as duas validações a seguir:

```
validates :email, :presence => true

validates_presence_of :email
```

Ambas possuem o mesmo comportamento, porém a sintaxe usando a *class macro* `validates` foca no atributo a ser validado, de forma que você possa adicionar diversas validações de uma vez. O `validates_presence_of`, em contrapartida, foca na validação em si, podendo passar diversos atributos de uma vez. Veja outro exemplo:

```
validates :email, :presence => true,
                  :format => { :with => /\A[^@]+@[^@\.]+\z/ },
                  :uniqueness => true
```

Ainda é possível colocar múltiplos atributos na validação:

```
validates :email, :full_name, :location, :presence => true
```

Como não há diferença de comportamento, você pode converter as validações que fizemos com a sintaxe tradicional para a sintaxe alternativa e verificar qual você mais gosta.

Nosso modelo usuário está bom o suficiente por enquanto. Vamos voltar a trabalhar nessa classe logo, porém vamos começar a fazer o fluxo de cadastro.

Capítulo 6

Tratando as requisições Web

Eu não desanimo, pois cada tentativa incorreta descartada é mais um passo a frente.
– Thomas A. Edison

6.1 Roteie as requisições para o controle

Agora que nosso modelo está pronto, vamos começar a ligar as outras partes do sistema. Vamos fazer tudo sem uso de geradores do Rails, para entendermos cada passo do projeto, diferente do recurso quarto.

Antes de tudo, vamos criar a rota, para que possamos testar as páginas e o controle conforme construímos. Lembra-se das sete rotas do Rails?

- **index** - Lista todos as entradas;
- **show** - Exibe uma entrada específica do recurso;
- **new** - Página para criar uma nova entrada;

- **create** - Ação de criar uma nova entrada;
- **edit** - Página para editar uma entrada;
- **update** - Ação de atualização de uma entrada existente;
- **destroy** - Remoção de uma entrada existente;

Agora, abra o arquivo `config/routes.rb`, adicione uma nova linha e teremos as ações prontas para nosso uso:

```
Colchonet::Application.routes.draw do
  resources :rooms
  resources :users

  # Aqui estarão vários comentários do Rails para
  # te ajudar a lembrar e entender como funciona
  # o roteador. Se você quiser, pode deletar tudo.
end
```

Agora, com o servidor do Rails em execução (se não estiver, basta executar `rails server` na pasta do projeto), abra o browser e acesse o endereço para o nosso recurso: http://localhost:3000/users/new.

Você vai esbarrar com uma tela de erro:

Routing Error

`uninitialized constant UsersController`

Try running `rake routes` for more information on available routes.

Figura 6.1: Erro: Routing error - Erro de roteamento

O *Routing error* (erro de roteamento) é um erro comum. O roteador do Rails tenta passar a execução de requisições de um recurso para o seu controle correspondente, que ainda não existe. O Rails consegue, através do nome do recurso,

derivar o nome do controle, de maneira que o recurso `:users` é mapeado para o `UsersController`. No caso do `:rooms`, o Rails mapeia para `RoomsController`. Isso é bastante conveniente e mais uma amostra do conceito de Convenção sobre Configuração.

Vá para a pasta `app/controllers` e crie o arquivo `users_controller.rb` de acordo com o seguinte:

```
class UsersController < ApplicationController
end
```

Essa classe deve herdar do `ApplicationController` que apesar do nome, não possui e não deve possuir uma rota para ela. Ela serve para que você configure todos os controles de sua aplicação e é um ponto de partida, portanto vamos alterá-la com o decorrer do projeto. Veja atualmente como é esta classe, abrindo o arquivo `app/controllers/application_controller.rb`:

```
class ApplicationController < ActionController::Base
  protect_from_forgery
end
```

O `ApplicationController` herda, por sua vez, do `ActionController::Base`, um componente do Rails. Nessa classe há uma *class macro* bastante importante, `protect_from_forgery`. Essa macro faz com que todos os controles da aplicação exijam uma chave de autenticação em ações de alteração de dados (*create*, *update* e *destroy*) de modo a evitar ataques de falsificação de requisição (*Request Forgery*). Portanto, é bastante importante deixá-la ativada sempre que possível.

Falsificação de requisição

Ataques de falsificação de requisição entre sites (ou *cross-site request forgery* - CSRF) consistem em enviar uma requisição de um site a outro fazendo uma ação que o usuário não desejou. Por exemplo, imagine que no colcho.net, para você deletar sua conta, basta você acessar http://colcho.net/usuarios/deletar_conta (endereço hipotético). Um usuário malicioso pode colocar em um site uma imagem com a seguinte tag:

Ao entrar nesse site, o browser irá tentar baixar o conteúdo dessa imagem, que na verdade é a ação para deletar sua conta e **bum!**, sua conta foi deletada. Para proteger seu site contra isso, o Rails faz um mecanismo tal que, para cada ação de modificação de estado, é necessário enviar uma chave de segurança gerada aleatoriamente.

Voltando ao projeto, vamos criar a nossa ação `new`, no `UsersController` (`app/controllers/users_controller.rb`):

```
class UsersController < ApplicationController
  def new
  end
end
```

Por enquanto não vamos fazer nada, dessa forma, o método `new` ficará vazio.

Ao abrir novamente a página, um erro diferente: **Template is missing**, ou "template ausente". A ação é encontrada, porém o Rails não consegue encontrar a apresentação daquela ação:

Template is missing

Missing template users/new, application/new with {:locale=>[:"pt-BR"], :formats=>[:html], :handlers=>[:erb, :builder, :coffee]}. Searched in: * "/Users/vinibaggio/Projects/book/rubyonrails/code/colchonet/app/views"

Figura 6.2: Erro: Template is missing - template ausente

Os templates ficam na pasta `app/views`. Dentro dela, existe uma hierarquia. Lembra-se como o Rails identifica o controle baseado no nome do recurso? O mesmo acontece aqui: todos os templates para um controle ficam em uma pasta de mesmo nome do recurso, ou seja, para o `UsersController`, os templates ficam na pasta `app/views/users`.

Na pasta `app/views` existe também a pasta `layouts`. Essa pasta é uma pasta especial. Nela ficam os layouts dos templates, ou seja, a base onde os templates serão colocados.

Observe a figura 6.3. Nela é possível ver como o Rails compõe uma página final, baseando-se em um layout e um template de uma ação.

```
<!DOCTYPE html>            app/views/layouts/application.html.erb
<html>
<head>
  <title>Colchonet</title>
  <%= stylesheet_link_tag    "application", :media => "all" %>
  <%= javascript_include_tag "application" %>
  <%= csrf_meta_tags %>
</head>
<body>

                               app/views/users/new.html.erb
  ...

</body>
</html>
```

Figura 6.3: Composição de página usando layouts

O uso de layouts dessa forma é bastante prático. Não há a necessidade de repetir diversos comandos e tags (evitando também um tedioso trabalho de busca/substituição quando você faz alguma alteração), basta escrever o HTML específico da sua página.

Para que um template se comporte como layout, basta executar `yield`, a mesma palavra-chave para executar blocos associados a métodos. Veja o exemplo a seguir:

`application.html.erb`:

```
<html>
  <body>
    <%= yield %>
  </body>
</html>
```

Neste exemplo, quando usarmos o layout `application`, todo o conteúdo do template será inserido no lugar do `yield`.

É possível também ter mais de um layout em um sistema. Isso é útil, por exemplo, quando há diversas "personas" em um site, como um administrador e um usuário comum, ou um lojista e o cliente.

Personas

Persona é um personagem de um possível usuário do seu site. É importante pensarmos em como cada persona que quisermos focar vai interagir com o sistema. Assim, é mais fácil ter ideias de como melhorar a experiência do usuário.

No caso do colcho.net, temos duas personas: o "hospedado", usuário que procura lugar para dormir e o "anfitrião", usuário que tem um lugar sobrando.

Vamos agora criar o template para a ação *new*. Para isso, crie a pasta `users` em `app/views` e crie o arquivo `app/views/users/new.html.erb`. O nome possui três significados: ação *new*, que vai ser pré-processado pelo ERB, resultando em um arquivo `html`:

```
<ul></ul>
```

Por enquanto esse template está muito sem graça. Vamos usar o controle `UsersController` para preparar os dados para que possamos criar o formulário de cadastro.

6.2 Integre o controle e a apresentação

Vamos agora trabalhar no `UsersController` para criar a tela de cadastro de novo usuário:

```
class UsersController < ApplicationController
  def new
    @user = User.new
  end
end
```

Duas coisas acontecem no trecho de código anterior: a primeira é que usamos o método `.new` no modelo `User`. Já vimos esse método anteriormente, ele cria uma nova instância do modelo `User`. Em seguida, armazenamos o novo objeto em uma variável de instância, `@user`. O controle compartilha todas as variáveis de instância com o template, ou seja, toda variável com `@` estará disponível em nossos templates.

Isso significa que já podemos acessar esse objeto no template. Usaremos ele para construir o formulário. Abra o arquivo `app/views/users/new.html.erb` e altere para que ele fique da seguinte forma:

```
<h1>Cadastro</h1>

<%= form_for @user do |f| %>
  <p>
    <%= f.label :full_name %><br />
    <%= f.text_field :full_name %>
  </p>
  <p>
    <%= f.label :location %><br />
    <%= f.text_field :location %>
  </p>
  <p>
    <%= f.label :email %><br />
    <%= f.text_field :email %>
  </p>
  <p>
```

```
    <%= f.label :password %><br />
    <%= f.password_field :password %>
  </p>
  <p>
    <%= f.label :password_confirmation %><br />
    <%= f.password_field :password_confirmation %>
  </p>
  <p>
    <%= f.label :bio %><br />
    <%= f.text_area :bio %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>
```

Esse código que criamos foi um exemplo de como é um template ERB: trechos de HTML envolto de código Ruby. Em ERB, todo código Ruby que não terá seu conteúdo impresso no HTML resultante deve ser envolto de <% %>. Caso você queira que o código seja impresso no HTML, temos que envolver o código Ruby com <%= %>.

Você pode ver também que este código faz uso extensivo de métodos auxiliares do Rails. Nesse caso, utilizamos alguns para construir páginas, em especial formulários (portanto devem ser chamados no objeto `f`), que recebem o nome do campo do objeto que estamos editando:

- `form_for` - Inicia a construção de um formulário para um modelo;
- `label` - Label correspondente a um campo do modelo;
- `text_field` - Campo simples de texto;
- `password_field` - Campo mascarado de senha;
- `text_area` - Área grande de texto, para escrevermos a bio;
- `submit` - Botão de Submit (Enviar), para enviar os dados ao servidor;

Você pode ter notado que em nenhum lugar nós mencionamos URLs para onde este formulário deve enviar os dados. Isso é mais uma vantagem de se usar a modelagem de recursos do Rails. O Rails é capaz de saber o recurso a que este formulário

pertence. E mais, através do uso do método `#new_record?` (experimente no console, execute em objetos gravados no banco e objetos novos), ele é capaz de saber se o formulário deve enviar à ação `create` ou `update`. Esperto, não?

Abra o browser na página http://localhost:3000/users/new:

Figura 6.4: Formulário de cadastro de usuário

O problema é que, ao preencher o formulário e clicar em "Create User" (criar usuário), vamos ser apresentados ao erro *Unknown action*, ou ação desconhecida, *create*. Isso significa que o formulário enviou os dados para uma ação que ainda não existe. Pois bem, vamos criá-la: abra novamente o `UsersController` (`app/controllers/users_controller.rb`):

```
class UsersController < ApplicationController
  def new
    @user = User.new
  end

  def create
    @user = User.new(params[:user])
    if @user.save
      redirect_to @user,
                  :notice => 'Cadastro criado com sucesso!'
    else
      render :new
    end
  end
end
```

A ação *create* possui a seguinte lógica:

- Cria novo usuário baseado nos dados de *parâmetros*;
- Se o usuário foi salvo com sucesso (`#save` retorna sempre `true` ou `false`), redireciona ele para a página de seu mais novo perfil e exibe uma mensagem felicitando o fato;
- Caso não seja possível salvar por causa de uma validação, renderiza o formulário novamente.

Vamos entrar em detalhe em cada linha:

```
@user = User.new(params[:user])
```

O método `params` (lembre-se que em Ruby, para chamar métodos, não precisamos de parênteses) retorna um hash com todos os parâmetros enviados pelo usuário, seja via formulário (**POST**) ou via *query string* (parâmetros pela URL, por exemplo http://google.com?q=hello, via **GET**). Quando usamos os métodos auxiliares do Rails para formulários, o `params` se parece com o seguinte:

```
{
  "authenticity_token"=>"ofiNLJQUjL/p5vX8z0cy+N5aE9htJDIAUk=",
  "user"=> {
    "full_name"=>"Vinicius",
    "location"=>"San Francisco, CA",
```

```
    "email"=>"vinibaggio@example.com",
    "password"=>"segredo",
    "password_confirmation"=>"segredo",
    "bio"=>"Olá, tudo bom? Meu nome é Vinícius."
  },
  "commit"=>"Create User",
  "action"=>"create",
  "controller"=>"users"
}
```

Você pode ver que todos os campos do formulário vieram dentro de uma hash só, a hash `users`. É esse conteúdo que passamos para o método `.new` do modelo.

Chave da hash em string ou símbolo?

Algumas coisas do Rails podem ser complicadas de serem entendidas para iniciantes, e uma delas é o hash `params`, que é um tipo de hash usado pelo Rails, chamado `HashWithIndifferentAccess`, ou hash com acesso indiferente. Com ela, você pode acessar chaves com símbolos ou strings, que no final dão no mesmo resultado:

```
puts params['user']
{
  "full_name"=>"Vinicius",
  "location"=>"San Francisco, CA"
}

puts params[:user]
{
  "full_name"=>"Vinicius",
  "location"=>"San Francisco, CA"
}
```

O método `.new` aceita um hash contendo todos os atributos a serem associados. Esse é um conceito chamado *mass-assignment*, ou seja, associação em massa. É muito mais prático do que ficar chamando cada método individualmente dessa maneira.

Porém nada nessa vida é de graça e, portanto, teremos alguns problemas que vamos ver ainda nesse capítulo.

Com o objeto todo populado de atributos que vieram do formulário, tentaremos salvá-lo no banco de dados. Se tudo der certo, as linhas a seguir são executadas:

```
redirect_to @user,
          :notice => 'Cadastro criado com sucesso!'
```

O método `redirect_to` envia ao browser do usuário um código de resposta "302" que significa "*Moved Temporarily*", dizendo ao navegador que ele deve ir para outro endereço. No conteúdo da resposta, uma localização é informada, causando o redirecionamento do browser ao novo endereço que é automaticamente determinado pelo uso do recurso (`@user`) como parâmetro. Nesse caso, o Rails irá redirecionar para a ação *show* do objeto `@user`.

O segundo parâmetro do método `redirect_to` é um hash de opções. Nesse caso, estamos usando um atalho para escrever uma mensagem via `flash`.

O `flash` é fundamentalmente um hash, que nesse caso estamos escrevendo na chave `:notice`. Portanto a linha poderia ser reescrita da seguinte forma:

```
flash[:notice] = 'Cadastro criado com sucesso!'
redirect_to @user
```

Uma característica importante do `flash` é que seu conteúdo é guardado na sessão do usuário e o seu conteúdo é eliminado em toda requisição. Quando você escreve no `flash`, o conteúdo só estará disponível na próxima requisição. Por isso, é normalmente utilizado em conjunto com redireções.

Finalmente, se o objeto não pôde ser salvo, executamos:

```
render :new
```

Isso porquê a ação *create* não possui um template/apresentação para si, então reaproveitamos o template do *new* para apresentar novamente o formulário e o usuário ter a oportunidade de corrigir os dados.

> **REDIRECT_TO OU RENDER?**
>
> É comum em ações *create* e *update* redirecionar o usuário para uma outra página quando tudo está certo. Porém, no caso de erros no objeto não é recomendado redirecionar. A razão é que, ao redirecionar, perdemos todos os parâmetros que o usuário enviou e também as mensagens de erro de validação quando executamos o método `#save`. Para essas situações, devemos usar o `#render`, que não causa o redirecionamento.

6.3 Controle o *mass-assignment*

Agora que entendemos bem o que a ação *create* faz, vamos executá-la! Vá ao formulário de cadastro (http://localhost:3000/users/new) e preencha o formulário. Ao clicar em 'Create User'...

```
Can't mass-assign protected attributes: password_confirmation
```

Erro de *mass-assignment*. O problema que estamos encontrando agora é na verdade uma proteção para o seu site. Imagine que temos o campo `admin` no modelo `User`, que é um campo booleano (ou seja, aceita apenas valores `true` ou `false`).

Mesmo sem o campo `admin` no formulário, um usuário mal intencionado pode forjar uma requisição da seguinte forma:

```
{
  "user" => {
    "full_name"=>"Pirata Malandro",
    "location"=>"Caribe",
    "email"=>"malandro@example.com",
    "password"=>"123",
    "password_confirmation"=>"123",
    "bio"=>"Rárr! Vou adquirir acesso de admin!",
    "admin"=>"1"
  }
}
```

Mesmo seu formulário não tendo o campo `admin`, o modelo `User` iria marcá-lo como `true` e pronto, facilmente um usuário pode forjar dados em seu sistema sem você perceber.

É por isso que, nas últimas versões do Rails, você é obrigado a criar uma *white-list*, ou seja, uma lista com os atributos que podem ser associados via *mass-assignment*. E, no nosso caso, o campo `password_confirmation` não está nessa lista. Abra o modelo `User` (`app/models/user.rb`):

```
class User < ActiveRecord::Base
  attr_accessible :bio, :email, :full_name, :location, :password

  validates_presence_of :email, :full_name, :location, :password
  validates_confirmation_of :password
  validates_length_of :bio, :minimum => 30, :allow_blank => false
  validates_format_of :email, :with => /\A[^@]+@([^@\.]+\.)+[^@\.]+\z/
end
```

Veja na segunda linha. A *class macro* `attr_accessible` (cuidado para não confundir com `attr_acessor`!) aceita uma lista de atributos que podem ser associados via *mass-assignment*. Vamos adicionar o atributo virtual `password_confirmation` à ela:

```
class User < ActiveRecord::Base
  # Cuidado para não esquecer a vírgula no final da linha.
  attr_accessible :bio, :email, :full_name, :location,
                  :password, :password_confirmation

  validates_presence_of :email, :full_name, :location, :password
  validates_confirmation_of :password
  validates_length_of :bio, :minimum => 30, :allow_blank => false
  validates_format_of :email, :with => /\A[^@]+@([^@\.]+\.)+[^@\.]+\z/
  validates_uniqueness_of :email
end
```

Depois dessa pequena alteração, reenvie o formulário, com todos os dados. Se não houveram erros no formulário, um novo erro será exibido: *Unknown action* (ação desconhecida), *show*. Isso significa que o objeto foi gravado com sucesso! Para comprovar, vá ao console do Rails:

```
# Busca o último objeto no banco de dados (via id)
User.last
#<User id: 1, full_name: "Vinicius", ...>
```

6.4 Exibição do perfil do usuário

Agora vamos criar a ação e o template de *show*, para não haver mais erros. Vamos primeiro ao controle `UsersController` (`app/controllers/users_controller.rb`):

```
class UsersController < ApplicationController
  def new
    @user = User.new
  end

  def show
    @user = User.find(params[:id])
  end

  def create
    @user = User.new(params[:user])
    if @user.save
      redirect_to @user, :notice => 'Cadastro criado com sucesso!'
    else
      render :new
    end
  end
end
```

Para a ação *show*, o código é simples: buscamos o objeto cujo id seja especificado pelo parâmetro `:id`. Esse parâmetro vem da rota, ou seja, se acessarmos a URL http://localhost:3000/users/1, o parâmetro `:id` será 1. Em seguida, usamos o método `.find` do modelo `User` para buscar exatamente aquele objeto. Associamos esse objeto à variável de instância `@user` para acessarmos no template.

E quando o id não existe?

Existe a possibilidade do usuário digitar um id não existente, ou um link estar incorreto. Quando isso acontecer, o `ActiveRecord` irá disparar uma exceção chamada `ActiveRecord::RecordNotFound`. Em desenvolvimento, vemos esse erro para ficar mais claro. Porém, o Rails sabe que não deve exibir esse erro ao usuário real, quando o sistema estiver em produção. Nesse caso, o Rails irá exibir a página de erro *404 Not Found*.

Outras exceções que resultam em erro *404 Not Found* em ambiente de produção são: `ActionController::RoutingError` (uma rota não existente) e `AbstractController::ActionNotFound` (rota existente mas ação não encontrada).

Vamos ao template. Crie o arquivo `app/views/users/show.html.erb` conforme o código a seguir:

```
<p id="notice"><%= notice %></p>

<h2>Perfil: <%= @user.full_name %></h2>

<ul>
  <li>Localização: <%= @user.location %></li>
  <li>Bio: <%= @user.bio %></li>
</ul>

<%= link_to 'Editar Perfil', edit_user_path(@user) %>
```

A estrutura desse template é um pouco diferente, pois estamos usando menos métodos auxiliares do Rails e mais HTML. Na verdade, a estrutura de templates de aplicações tendem a ser mais dessa forma, uma mistura de HTML com pinceladas de ERB e métodos auxiliares.

O método `notice` é um método auxiliar do Rails para retornar o conteúdo do `flash[:notice]`. Mas o método mais importante para nós neste momento é o método `link_to`.

O método `link_to` serve para gerar links, através de tags <a>. O primeiro atributo, como pode perceber, é o texto que vai no link, e o segundo é uma URL. O método `edit_user_path(@user)` é um método gerado pelo roteador de acordo com

seu arquivo de rotas. O método, por sua vez, vai gerar a rota correta para a ação de *edit* do usuário `@user`.

Com o template pronto, quando você criar um novo usuário, você verá a seguinte imagem:

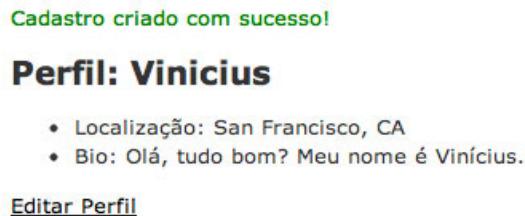



Figura 6.5: Perfil do usuário, depois de efetuar um cadastro

6.5 Permita a edição do perfil

A imagem anterior mostra que temos uma tela bem simples, mas já estamos começando a amarrar toda a funcionalidade de cadastro. Ainda faltam as ações de *edit* e *update*. Elas são bem parecidas com as ações *new* e *create*. Vamos primeiro à ação *edit*, no controle `UsersControllers` (`app/controllers/users_controller.rb`):

```
class UsersController < ApplicationController
  # Omitindo as outras ações para não atrapalhar...

  # new
  # show
  # create

  def edit
    @user = User.find(params[:id])
  end
end
```

Buscamos o objeto cujo ID vem do parâmetro, da mesma forma que a ação `show`. Fazemos isso para que o formulário venha populado com os dados que o usuário já preencheu anteriormente. Vamos ao template da ação `edit` (`app/views/users/edit.html.erb`):

```
<h1>Editar perfil</h1>

<%= form_for @user do |f| %>
  <p>
    <%= f.label :full_name %><br />
    <%= f.text_field :full_name %>
  </p>
  <p>
    <%= f.label :location %><br />
    <%= f.text_field :location %>
  </p>
  <p>
    <%= f.label :email %><br />
    <%= f.text_field :email %>
  </p>
  <p>
    <%= f.label :password %><br />
    <%= f.password_field :password %>
  </p>
  <p>
    <%= f.label :password_confirmation %><br />
    <%= f.password_field :password_confirmation %>
  </p>
  <p>
    <%= f.label :bio %><br />
    <%= f.text_area :bio %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>
```

6.6 Reaproveite as apresentações com partials

Você deve estar com uma grande sensação de *déjà vu* nesse momento. Outra sensação que você pode ter é desgosto por ter duas páginas tão parecidas, no bom e velho *copy'n'paste*, que estraga a vida de tantas pessoas. Mas não se desespere! Vamos resolver isso agora.

O componente de templates do Rails (`ActionView`) possui um recurso chamado *partials*, mini-templates que podem ser incluídos em templates, ou seja, uma *partial*

não é usada para renderizar diretamente uma ação, mas um template pode usar uma ou mais *partials* para compor o resultado final.

Para criar uma *partial*, basta criar um template com seu nome iniciando em `_`. Ou seja, nesse caso, vamos criar a *partial* "form". Portanto, crie o arquivo `_form.html.erb`, na pasta `app/views/users/`:

```
<%= form_for @user do |f| %>
  <p>
    <%= f.label :full_name %><br />
    <%= f.text_field :full_name %>
  </p>
  <p>
    <%= f.label :location %><br />
    <%= f.text_field :location %>
  </p>
  <p>
    <%= f.label :email %><br />
    <%= f.text_field :email %>
  </p>
  <p>
    <%= f.label :password %><br />
    <%= f.password_field :password %>
  </p>
  <p>
    <%= f.label :password_confirmation %><br />
    <%= f.password_field :password_confirmation %>
  </p>
  <p>
    <%= f.label :bio %><br />
    <%= f.text_area :bio %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>
```

Apenas um corte do título, o resto é mantido igual. Nas *partials* também temos acesso à variável `@user`. Como as ações *edit* e *new* usam a mesma variável, isso tornam as coisas bastante convenientes. Agora vamos aos templates `app/views/users/new.html.erb` e `app/views/users/edit.html.erb`:

`app/views/users/new.html.erb`:

```
<h1>Cadastro</h1>

<%= render 'form' %>
```

Note que o nome da *partial*, quando fazemos o render na página, não inclui o _. O Rails incluirá automaticamente, sabendo que é uma *partial*.

app/views/users/edit.html.erb:

```
<h1>Editar perfil</h1>

<%= render 'form' %>
```

Depois de editar os dois arquivos, salve-os e navegue nas páginas. Funciona perfeitamente, não? E qualquer modificação será refletida em ambas as páginas, evitando o *copy'n'paste*, o trabalho repetitivo e evitando erros.

Ok, agora que os templates estão melhorados, vamos implementar a ação *update*. Abra o controle UsersController novamente (app/controllers/users_controller.rb):

```
class UsersController < ApplicationController
  # Omitindo as outras ações para não atrapalhar...

  # new
  # show
  # create

  def edit
    @user = User.find(params[:id])
  end

  def update
    @user = User.find(params[:id])
    if @user.update_attributes(params[:user])
      redirect_to @user, :notice => 'Cadastro atualizado com sucesso!'
    else
      render :update
    end
  end
end
```

O funcionamento do *update* é quase o mesmo do *create*. A principal diferença é que agora fazemos a atualização de um objeto específico, identificado pelo parâmetro

`:id`, ao invés de criar um novo objeto na coleção. O método `#update_attributes` aceita um hash de atributos de maneira similar ao método `.new` para criar um novo objeto. Inclusive, o método `#update_attributes` também possui a proteção de *mass-assignment*.

Se a atualização do perfil ocorrer com sucesso, o usuário é redirecionado para a página do seu perfil, contendo uma mensagem felicitando o ocorrido, como é possível ver na figura 6.6

Cadastro atualizado com sucesso!

Perfil: Vinicius

- Localização: San Francisco, CA
- Bio: O meu perfil anterior estava desatualizado, por isso estou atualizando-o agora.

Editar Perfil

Figura 6.6: Atualização de perfil com sucesso

Se na atualização ocorrer algum erro, o formulário de edição será exibido, de forma que o usuário possa corrigir os dados.

6.7 Mostre os erros no formulário

Falando em erros, ao enviar um formulário que contenha informações preenchidas incorretamente, os campos ficam marcados com a cor vermelha. Isso acontece pois o Rails automaticamente adiciona classes ao campo do formulário quando há erro e, graças ao *scaffold* que fizemos no capítulo 4, temos um CSS básico que formata esse HTML.

Figura 6.7: Marcação de erros no formulário

O principal problema desse formulário é que ele não informa exatamente o erro que aconteceu para que o usuário possa corrigir o campo. Sem nenhuma informação, é praticamente impossível o usuário entender o que precisa ser alterado. Por isso, vamos adicionar essa dica.

Como a *partial* do formulário é exibida tanto na ação `edit` quanto na `new`, a *partial* torna-se uma excelente candidata a receber a exibição de erros. Por isso, abra o arquivo `app/views/users/_form.html.erb` e adicione o seguinte código no início do arquivo:

```
<% if @user.errors.any? %>
  <div id="error_explanation">
    <h2>Infelizmente não foi possível completar
    a ação pois o formulário possui os seguintes erros:</h2>
    <ul>
      <% @user.errors.full_messages.each do |message|%>
        <li><%= message %></li>
      <% end %>
    </ul>
  </div>
<% end %>
```

```
<%= form_for @user do |f| %>
  ...
<% end %>
```

O que esse código faz é verificar se há algum erro (o método `#any?` retorna `true` se pelo menos um elemento de uma `Array` é diferente de `nil`). Se houver um ou mais erros, cria-se a `div` com uma mensagem e em seguida lista todas as mensagens de erro. O resultado é o seguinte:

Figura 6.8: Mensagem de erro com mensagem amigável

Agora o usuário pode ler a mensagem e tomar uma ação para corrigir seus erros. O funcionamento das páginas está correto, porém vamos adicionar links para melhorar a navegação.

6.8 Configure a ação raiz (root)

A ação raiz, ou *root*, é a ação que é executada quando vamos ao endereço raiz do site, por exemplo, http://www.colcho.net, ou http://localhost:3000/. O que temos agora

é ainda a página de boas vindas do Rails. Vamos mudar isso.

Primeiro, remova o arquivo `public/index.html`, caso contrário o Rails sempre irá responder com essa página, ignorando seu arquivo de rotas. Vamos editar o `routes.rb` para incluir a rota *root*:

```
Colchonet::Application.routes.draw do
  resources :rooms
  resources :users

  root :to => "home#index"
end
```

A notação de controles, quando mencionados manualmente, é o seu nome (por exemplo, "users" para o `UsersController`), sustenido (`#`, ou também 'jogo da velha') e o nome da ação. Dessa forma, "home#index" aponta para o controle `HomeController`, ação *index*. Crie o arquivo `app/controllers/home_controller.rb`:

```
class HomeController < ApplicationController
  def index
  end
end
```

> **BOA PRÁTICA: SEMPRE DECLARE TODAS AS AÇÕES**
>
> Quando uma ação não faz nada, é possível simplesmente não implementar nenhum método no controle e apenas criar o template. O `HomeController`, por exemplo, não necessitaria do método `index`, pois tem o template para exibir a página e não contém absolutamente nada.
>
> Porém, é sempre interessante declarar todos os métodos que o controle responde, para que fique claro ao leitor do código (que pode ser você mesmo depois de algumas alguns anos, lembre-se disso!).

Agora, crie a pasta `app/views/home` e o template `app/views/home/index.html.erb`:

```
<h1>Colcho.net</h1>
```

```
<p>
  Escolha uma das ações a seguir:
  <ul>
    <li><%= link_to "Cadastro", new_user_path %></li>
    <li><%= link_to "Visualizar quartos", rooms_path %></li>
  </ul>
</p>
```

Agora sim, temos links para as principais páginas. Vamos adicionar dois novos links, para que o usuário possa desistir de cadastrar ou atualizar o seu perfil:

`app/views/users/new.html.erb`:

```
<h1>Cadastro</h1>

<%= render 'form' %>

<%= link_to "Voltar", root_path %>
```

`app/views/users/edit.html.erb`:

```
<h1>Cadastro</h1>

<%= render 'form' %>

<%= link_to "Voltar", @user %>
```

Note que, como as rotas são diferentes, não colocamos na *partial*, mas no template em si. No `edit` usamos a capacidade do Rails de determinar a rota de `show` de um recurso, apenas colocando o objeto como parâmetro de URL.

Por fim, vamos editar a página do perfil do usuário, `app/views/users/show.html.erb`, adicionando um link ao final:

```
...

<%= link_to 'Editar Perfil', edit_user_path(@user) %>
<%= link_to 'Home', root_path %>
```

A rota `root_path` é especial, e aponta para a raiz do site. Pronto, agora temos interligação entre todas as páginas do site, inclusive à área de quartos que deixamos intocado por enquanto. Chegaremos lá.

Capítulo 7

Melhore o projeto

Se longe enxerguei foi porque me apoiei no ombro de gigantes.
– Isaac Newton

7.1 Lição obrigatória: sempre aplique criptografia para armazenar senhas

Se você abrir o console do Rails e procurar pelos cadastros, vai poder observar que é possível ler as senhas dos seus usuários e isso não é nem um pouco profissional.

```
# Senha nada secreta...
User.first
# <User id      : 1,
#   full_name  : "Vinicius",
#   email      : "vinibaggio@example.com",
#   password   : "segredo",
#   location   : "San Francisco, CA",
#   bio        : "O meu perfil anterior...",
```

```
#   created_at : "2012-06-21 05:37:34",
#   updated_at : "2012-06-22 06 : 36 : 59">
```

Vamos usar uma funcionalidade do Rails para guardar as senhas usando encriptação BCrypt, chamada `has_secure_password`. Para isso, vamos precisar mudar a coluna `password` no banco de dados para `password_digest` e mudar algumas coisas no modelo.

> **Encriptação de senha**
>
> O algoritmo usado pelo Rails para encriptar senhas é o BCrypt, bastante eficiente e extremamente recomendado. Nesse algoritmo, a senha em texto puro passa por diversos cálculos matemáticos que a torna totalmente cifrada e é um processo irreversível (por isso o nome "digest").
>
> Para verificar se uma senha é válida, devemos fazer os mesmos cálculos com a senha que o usuário fornecer e comparar com o resultado final que tivermos no banco de dados. Se forem iguais, o usuário digitou a senha corretamente. Se alguém obter acesso ao seu banco de dados, será impossível reverter o processo e obter o texto original.
>
> Este é um processo completamente seguro, se o algoritmo utilizado for forte, como é o BCrypt. Antigamente usava-se algoritmos como MD5 e SHA1, mas já foram provados serem ruins para esse fim, pois há muita colisão (strings diferentes que resultam no mesmo "digest"), além de outros fatores.

7.2 Como adicionar plugins ao projeto?

Para que o Rails consiga usar o algoritmo BCrypt, precisamos fazer com que o Rails use a gem chamada `bcrypt-ruby`. Para isso, abra o arquivo `Gemfile` na raiz do seu projeto e descomente a linha mencionando a gem:

```
# To use ActiveModel has_secure_password
gem 'bcrypt-ruby', '~> 3.0.0'
```

Em seguida, usando o terminal, vá à raiz do seu projeto e execute o comando `bundle`:

```
$ bundle
Using rake (0.9.2.2)
Using i18n (0.6.0)
Using multi_json (1.3.6)
Using activesupport (3.2.8)
Using builder (3.0.0)
...
Using activeresource (3.2.8)
Installing bcrypt-ruby (3.0.1) with native extensions
Using bundler (1.1.4)
...
Your bundle is complete!
```

Esse comando faz com que a gem seja baixada e instalada em seu sistema. O arquivo `Gemfile` e o comando `bundle` fazem parte de uma ferramenta chamada Bundler (http://gembundler.com/). Ela é uma ferramenta complexa e bastante poderosa de gerenciamento de dependências dos seus aplicativos. O Rails usa-o para carregar todas as bibliotecas com as versões corretas. O trabalho pesado fica para a ferramenta, para nós programadores nos resta manter o arquivo `Gemfile` organizado.

Após a execução do `bundle`, o bundler irá gerar uma nova versão do `Gemfile.lock`, que contém exatamente todas as dependências e versões de gems que satisfazem as dependências da aplicação Rails. Se você tiver curiosidade, pode ver as dependências de uma gem vendo o conteúdo desse arquivo.

Reiniciando o console e o servidor

Em mudanças fundamentais como essa, é necessário reiniciar tanto o console quanto o servidor, mesmo com o uso do `reload!`. Isso deve-se ao fato de que o Rails carrega suas dependências no momento que é iniciado e o novo código sendo carregado faz referências às bibliotecas que o Rails não carregou anteriormente.

7.3 Migração da tabela users

Uma vez a gem instalada, é necessário satisfazer a segunda exigência do `has_secure_password`, que é ter a coluna `password_digest`. Para isso, vamos gerar uma nova migração. Execute, na pasta raiz do projeto:

```
$ rails generate migration rename_password_on_users
  invoke  active_record
  create    db/migrate/20120623053658_rename_password_on_users.rb
```

Com a migração gerada, abra o arquivo recém criado. O nome do arquivo irá variar conforme a hora que a migração foi gerada, mas você pode usar a saída do comando de geração da migração para saber exatamente qual arquivo editar.

Você terá o seguinte:

```
class RenamePasswordOnUsers < ActiveRecord::Migration
  def up
  end

  def down
  end
end
```

Como nossa mudança é simples, não é necessário especificar o código para `#up` e `#down` separadamente, vamos usar o `#change`:

```
class RenamePasswordOnUsers < ActiveRecord::Migration
  def change
    rename_column :users, :password, :password_digest
  end
end
```

Agora basta executar a migração com a tarefa rake `rake db:migrate`:

```
$ rake db:migrate
==  RenamePasswordOnUsers: migrating ====================
-- rename_column(:users, :password, :password_digest)
   -> 0.0084s
==  RenamePasswordOnUsers: migrated (0.0085s) ===========
```

Por que não fizemos certo desde o início?

Você deve estar se perguntando... Por que não fizemos certo desde o início? A resposta é simples. É raro o dia que sabemos toda a modelagem do nosso sistema logo de início. É quase certo que um dia você precisará alterar o modelo por causa de alguma decisão tomada ou algum conhecimento adquirido. O Rails dá ferramentas para dar suporte a esse tipo de desenvolvimento, ao invés de gastar dias e dias com modelagem que no final pode não ser necessária. O mesmo acontece com o colcho.net!

Migração feita, agora é hora de alterar o modelo. Segundo a documentação do `has_secure_password` (procure por `has_secure_password` no site http://api.rubyonrails.org, se tiver interesse), essa *class macro* já nos dá as validações de confirmação de senha e a presença de senha. Além disso, ela cria dois novos atributos virtuais, o `password` e `password_confirmation`. Portanto, o que devemos fazer no model `User` é:

- Remover a validação de presença de senha;
- Remover a validação de confirmação de senha;
- Adicionar a *class macro* `has_secure_password`

Contudo, não é necessário alterar nenhum formulário, já que a ainda usaremos o campo `password` nos formulários. Portanto a única alteração é o modelo `User` (`app/models/user.rb`), que deverá ficar assim:

```
class User < ActiveRecord::Base
  attr_accessible :bio, :email, :full_name, :location, :password,
                  :password_confirmation

  validates_presence_of :email, :full_name, :location
  validates_length_of :bio, :minimum => 30, :allow_blank => false
  validates_format_of :email, :with => /\A[^@]+@([^@\.]+\.)+[^@\.]+\z/
  validates_uniqueness_of :email

  has_secure_password
end
```

Pronto, tudo terminado. Você pode usar a interface para atualizar a senha (ainda não temos controle de permissão, então não temos que nos preocupar por enquanto). Vá ao console e verifique a sua senha, bem diferente, não?

```
User.first.password
# => nil

User.first.password_digest
# => "$2a$10$jvaS/caj8ohAkBG8.iJEqeFVRA4xw/XlJv/NF9NZrZmsbPO9/S4GK"
```

Para verificar a senha, basta usar o método `authenticate`, também criado automaticamente pela *class macro* `has_secure_password`, que irá retornar false quando a senha estiver inválida e o próprio objeto quando estiver correta:

```
vinicius = User.first
# => #<User id: 1, ...>

vinicius.authenticate 'invalido'
# => false

vinicius.authenticate 'segredo'
# => #<User id: 1, ...>
```

Bem simples não? Vamos usar esse mecanismo no capítulo 10 com mais detalhes para fazer o controle do login do usuário. Seus usuários agora possuem senhas seguras. Agora, precisamos dar um estilo aos templates, eles estão muito ruins!

7.4 Melhoria de templates e CSS

Até agora estamos usando o CSS gerado pelo *scaffold* do Rails. Embora seja funcional, ele é bem cru:

Figura 7.1: Site com o layout do scaffold

Os formulários são bem agressivos visualmente:

Editar perfil

Infelizmente não foi possível completar a ação pois o formulário possui os seguintes erros:

- Location can't be blank

Full name
Vinicius

Location

Email
vinibaggio@gmail.com

Password

Password confirmation

Figura 7.2: Formulário com erro

Com um pouco de HTML e CSS, vamos deixar o site um pouco mais agradável de ser visto:

Figura 7.3: Nova home

Vamos também melhorar a exibição de erros para que as mensagens fiquem lado

a lado com os campos a serem corrigidos:

Figura 7.4: Formulário com melhor notificação de erros

Para fazer tudo isso, vamos trabalhar bastante na camada de apresentação do site, alterando templates, alterando CSS e criando métodos auxiliares para tornar o template mais simples.

Conhecimentos de HTML e CSS

Infelizmente não dá para explicar detalhadamente cada parte do que vamos fazer em seguida, pois este não é o foco do livro. Se você gostaria de aprender mais sobre esses assuntos (recomendado se você quiser se tornar um profissional completo de web!), recomendo fortemente a leitura do livro *HTML5 e CSS 3: Domine a web do futuro*, escrito pelo Lucas Mazza.

7.5 Trabalhe com layout e templates para melhorar sua apresentação

O primeiro passo é alterar o layout da aplicação para incluir o cabeçalho com o menu na direita. Para isso, abra o arquivo `app/views/layouts/application.html.erb` e coloque o seguinte conteúdo:

```
<!DOCTYPE html>
<html>
<head>
  <title>Colchonet</title>
  <link href='http://fonts.googleapis.com/css?family=Pacifico'
                    rel='stylesheet' type='text/css'>
  <%= stylesheet_link_tag    "application", :media => "all" %>
  <%= javascript_include_tag "application" %>
  <%= csrf_meta_tags %>
</head>
<body>
  <header>
    <div id="header-wrap">
      <h1><%= link_to "colcho.net", root_path %></h1>
      <nav>
        <ul>
          <li><%= link_to "Quartos", rooms_path %></li>
          <li><%= link_to "Cadastro", new_user_path %></li>
        </ul>
      </nav>
    </div>
  </header>
```

```
  <div id="content">
    <% if notice.present? %>
      <p id="notice"><%= notice %></p>
    <% end %>
    <% if alert.present? %>
      <p id="alert"><%= alert %></p>
    <% end %>

    <%= yield %>
  </div>
</body>
</html>
```

As modificações nesse template foram:

- A inclusão do CSS para o uso da fonte "Pacifico", disponível no Google Web Fonts para o nosso logo (não se preocupe se você não tiver acesso à internet, vamos usar fontes *fallback*);
- A criação da tag `header` e seu conteúdo, o logotipo e o menu de navegação;
- Embrulhar o `yield` em uma tag `content`, para centralizar todo o conteúdo dos templates.

Antes de continuar, vamos tirar a referência ao `flash` da ação `show` do usuário e alterar a tag `h2` para `h1`:

```
<h1>Perfil: <%= @user.full_name %></h1>

<ul>
  <li>Localização: <%= @user.location %></li>
  <li>Bio: <%= @user.bio %></li>
</ul>

<%= link_to 'Editar Perfil', edit_user_path(@user) %>
```

Como agora temos o nome do site e o menu em todas as páginas, aquela página inicial (ou "home") que criamos na seção 6.2 fica redundante. Vamos alterá-la para mostrar ao usuário até no máximo três quartos que já foram cadastrados.

Antes de alterar o template, porém, precisamos alterar o controle `HomeController`. Edite o arquivo `app/controllers/home_controller.rb`:

```
class HomeController < ApplicationController
  def index
    @rooms = Room.limit(3)
  end
end
```

O método `.limit` é um método que faz uma busca no modelo `Room`, limitando a três resultados. A ordem depende do banco de dados, pois a `query` SQL resultante é:

```
SELECT "rooms".* FROM "rooms" LIMIT 3
```

Por enquanto isso é suficiente. Agora vamos editar o template do controle `HomeController`, ação *index* (arquivo `app/views/home/index.html.erb`):

```
<h1>Quartos recém postados</h1>
<ul>
  <% @rooms.each do |room| %>
    <li><%= link_to "#{room.title}, #{room.location}", room %></li>
  <% end %>
</ul>
```

Se você fizer um *refresh* na página inicial, já vai observar a inclusão desses quartos. Mas antes de continuarmos, vamos dar uma melhorada no código anterior.

Você pode estar se perguntando: "qual o problema desse código?". E a resposta é simples: esse é um típico caso em que estamos violando uma regra de negócio que parece inofensiva: o nome completo de um quarto. Mas em casos como esse é comum que os programadores acabem repetindo a composição diversas vezes em templates espalhados. E o dia que você mudar essa lógica, vai se perder em diversos "search and replaces". Para isso não acontecer, vamos fazer do jeito certo: o template deve saber o **mínimo** possível de como o modelo é feito.

Abra agora o então intocado modelo `Room` (`app/models/room.rb`) e adicione o método `#complete_name`:

```
class Room < ActiveRecord::Base
  attr_accessible :description, :location, :title

  def complete_name
    "#{title}, #{location}"
  end
end
```

Voltando ao template (`app/views/home/index.html.erb`):

```
<h1>Quartos recém postados</h1>
<ul>
  <% @rooms.each do |room| %>
    <li><%= link_to room.complete_name, room %></li>
  <% end %>
</ul>
```

O resultado visual é o mesmo, mas o código do template está mais limpo, então podemos continuar.

7.6 O que é o Asset Pipeline?

Antes de começarmos a modificar os *stylesheets* (CSSs) da aplicação, vamos parar um pouco e entender mais sobre um dos componentes mais controversos, em minha opinião, do Rails.

O *Asset Pipeline* vem para resolver um problema complicado da web moderna: a entrega de *assets*. O que significa isso? Não seria simplesmente entregar um arquivo javascript ou stylesheets ao browser do usuário?

A resposta não é tão simples. Como as aplicações web tem se tornado complexas, diversas ferramentas estão sendo utilizadas para tornar o desenvolvimento de javascripts e *stylesheets* mais produtivo. Além disso, é necessário diminuir ao máximo o tempo que o browser leva para baixar esses arquivos, para que as aplicações web não passem a percepção de serem lentas.

Dessa forma, hoje em dia, é comum que aplicações web possuam uma ou mais das seguintes fases para a entrega de um *asset*:

1) Pré-compilação: Transformação de um SCSS ou LESS em CSS puro, ou CoffeeScript em JavaScript;

2) Concatenação: Juntar todos os arquivos CSS em um único `.css`, ou todos os arquivos JavaScript em um único `.js`;

3) "Minificação": usar técnicas para diminuir o tamanho dos arquivos, como renomear variáveis (de "essaVariavelEhGrande" para "a");

4) Compressão: Usar compressão GZip para diminuir ainda mais o tamanho do arquivo, caso o browser tenha suporte (a maioria dos browsers modernos possui suporte a GZip).

> **SCSS e CoffeeScript são opcionais**
>
> Por padrão, o Rails já instala SASS (e por consequência, SCSS) e CoffeeScript em seu projeto. Porém, o uso de SCSS e CoffeeScript é opcional, ou seja, você pode continuar escrevendo JavaScript e CSS puro e ainda aproveitar dos outros benefícios do Assets Pipeline.

Em conjunto com tudo isso, existe outra preocupação bastante importante, que é o *caching*. Além do seu browser, existem diversos elementos de rede que podem fazer *caching* de *assets* (arquivos estáticos), como seu provedor de internet, *firewalls*, CDNs (*Content Delivery Network*, ou rede de entrega de conteúdo), equipamentos de rede, entre outros. O problema é que, quando você atualiza um asset no servidor, o conteúdo desse arquivo pode estar em *cache* e portanto seus usuários podem ter problemas ao acessar o site.

Todos os problemas e processos são bastante complicados, e é o objetivo do *Asset Pipeline* tornar isso tudo o mais simples possível. Como isso é feito?

Primeiro, o Rails, através de uma biblioteca chamada `sprockets`, lê cada manifesto da sua aplicação. Um manifesto nada mais é do que um arquivo que contém diretivas declarando quais são os arquivos que são dependência do seu aplicativo.

Com essa lista de arquivos, o Rails compila cada um de acordo com as extensões utilizadas. Por exemplo, se o nome do seu *stylesheet* for `users.css.scss.erb`, o *stylesheet* vai primeiro ser pré-processado em ERB e, em seguida, SASS (SASS e SCSS são pré-processados pelo próprio SASS), gerando um arquivo CSS final.

Após obter todos os arquivos pré-compilados, o Rails irá juntar todos os arquivos de um manifesto e gerará um arquivo cujo nome é o nome desse manifesto. Por exemplo, se o manifesto `application.css` mencionar os arquivos `footer.css` e `reset.css` e o manifesto `admin.css` mencionar apenas o `reset.css`, o resultado final desse processo serão dois únicos arquivos, o `application.css`, contendo o `footer.css` e o `reset.css` concatenados e o `admin.css`, apenas o `reset.css`.

No final, o browser do usuário terá que baixar muito menos arquivos, tornando a carga da página mais rápida.

Após a concatenação, o que acontece é a minificação. O Rails (ou mais especificamente, o `sprockets`) gera uma versão minificada de cada arquivo gerado no passo anterior, usando ferramentas bastante eficazes. O processo de minificação gera um

arquivo javascript ou *stylesheet* completamente válido, mas praticamente ilegível. O único objetivo desse passo é diminuir o tamanho do texto a ser baixado pelo usuário.

Chegando ao final, o Rails calcula o *digest* MD5 do arquivo e adiciona ao nome do arquivo. Por exemplo, se o arquivo chama-se `application.css`, ele se tornará algo do tipo:

```
application-e049a640704156e412f6ee79daabc7f6.css
```

Esse número gerado depende do conteúdo do arquivo, portanto se uma nova versão desse *asset* for gerada, esse número será alterado. Isso fará com que o mecanismo de *caching* não seja acionado caso uma nova versão do *asset* seja compilada.

Por fim, há a compressão desses *assets* com o algoritmo GZip, gerando:

```
application-e049a640704156e412f6ee79daabc7f6.css.gz
```

Com o suporte a GZip de servidores como Apache (http://httpd.apache.org/) ou nginx (pronuncia-se "engine x", ou ênginéx) (http://http://nginx.com/) e dos browsers, a entrega desses *assets* se torna muito mais eficiente e diminuindo o tempo de carga da página.

Note que, para esse mecanismo funcionar, é estritamente necessário que executemos a pré-compilação desses *assets* nos servidores:

```
rake assets:precompile
```

Isso irá gerar os arquivos de *assets*, que não serão entregues pelo Rails, mas sim pelo seu servidor web.

No modo de desenvolvimento de aplicações Rails, que é o modo que estamos executando, as coisas são um pouco mais simples. Ao invés de serem concatenados e compactados, os *assets* são apenas pré-compilados automaticamente por cada página.

Agora que entendemos um pouco do que é o Asset Pipeline, vamos usá-lo para criar *stylesheets* para o Colcho.net.

7.7 Criando os novos stylesheets

Abra o arquivo `app/assets/stylesheets/application.css`. Você pode ver que não há nenhum código CSS de fato, mas há um conjunto de comentários importantes que estão lá para compor o que chamamos de "manifesto", que são:

```
/*
 *= require_self
 *= require_tree .
 */
```

Esses dois comentários são diretivas do `sprockets` (note o "=" na linha). Eles possuem um significado importante. O `require_self` faz com que qualquer CSS que esteja no arquivo do manifesto seja incluído antes do restante, e o `require_tree` varre toda a árvore de diretórios a partir do diretório especificado (no caso, ".", ou pasta atual) e inclui os *stylesheets* no manifesto.

Isso nos diz em termos práticos que qualquer CSS que incluirmos na pasta `app/assets/stylesheets/` será incluído no manifesto final, e isso é suficiente para nós.

Não vamos mais utilizar o estilo gerado pelo *scaffold*, então vamos deletá-lo. Para isso, basta apagar o arquivo `app/assets/stylesheets/scaffolds.css.scss`. Você pode apagar também o CSS gerado pelo *scaffold* para o controle `RoomsController`: `app/assets/stylesheets/rooms.css.scss`.

Vamos começar a criar o nosso próprio *stylesheet*. Como a maioria dos browsers não possuem padrões razoáveis, é comum usar um *stylesheet* chamado "reset", que deixa todos os estilos em um mesmo padrão para que, a partir daí, possamos construir o nosso próprio.

Normalmente usamos o famoso "Eric Meyer's Reset" (http://meyerweb.com/eric/tools/css/reset/), porém não precisamos de toda a cobertura do "reset" do Meyer. Portanto, vamos criar o nosso próprio. Crie o arquivo `app/assets/stylesheets/reset.css`:

```
* {
  margin: 0;
  padding: 0;
  text-decoration: none;
}
li { list-style: none; }
```

Esse CSS simples deixa o `padding` e a `margin` em valores zerados. Agora vamos ao estilo do site em geral. Como o CSS possui algumas dezenas de linhas, vou mostrar pouco a pouco e você deve ir colocando cada trecho um após o outro.

Crie o arquivo `app/assets/stylesheets/default.css.scss`. Note que esse não é um arquivo de CSS comum e portando **deve** ter a extensão `.scss`. Isso significa que esse CSS é um SCSS, uma extensão de CSS que adiciona funcionalidades

como regras aninhadas, variáveis, *mixins* e outras várias coisas úteis. Para entender melhor como funciona, veja o site oficial do projeto SASS (http://sass-lang.com).

A primeira parte é a declaração de variáveis em SCSS, de forma que se quisermos alterar, por exemplo, a largura do conteúdo, mudamos em apenas um lugar e a alteração se reflete onde a variável estiver sendo usada:

```
$header-height: 55px;
$content-width: 700px;

$serif-families: "Pacifico", "Georgia", serif;
$sans-serif-families: "Helvetica", sans-serif;

$error-text-color: #B94A48;
$success-text-color: #468847;
```

Em seguida, declaramos o estilo geral da página, como cor do fundo, tamanho e família da fonte do texto do site (usando a variável `$sans-serif-families`):

```
* {
  font-family: $sans-serif-families;
  font-size: 14px;
}

body { background-color: #f5f5f5; }
```

Declaramos um *mixin* para facilitar a declaração de `box-shadow`, colocando todos os *vendor prefixes* quando necessário:

```
@mixin shadow($color, $x, $y, $radius) {
  -moz-box-shadow:    $color $x $y $radius;
  -webkit-box-shadow: $color $x $y $radius;
  box-shadow:         $color $x $y $radius;
}
```

Em seguida, vamos criar a barra de navegação, que fica no topo. Criamos a cor de fundo da barra e uma sombra na parte inferior. Criamos também o `#header_wrap`, que é responsável pela centralização do conteúdo da barra de navegação em conjunto com o conteúdo que ficará posteriormente. Por fim, estilizamos o logotipo. Note que vamos usar o *mixin* criado anteriormente via `@include`, evitando ter que repetir três regras para cada *vendor prefix*.

```
header {
  @include shadow(#ccc, 0, 3px, 6px);

  border-bottom: 1px solid #686;
  margin-bottom: 15px;

  #header-wrap {
    width: $content-width;
    margin: 0 auto;
  }

  height: $header-height;
  background-color: #9ECE71;
  h1 {
    float: left;
    a {
      color: #333;
      font-family: $serif-families;
      font-weight: 400;
      font-size: 2.5em;
      &:hover {
        color: #000;
      }
    }
  }
}
```

A segunda parte da barra superior, agora é o estilo do menu de navegação.

```
header nav {
  line-height: $header-height;
  float: right;
  li {
    display: inline;
    background-color: #546f3c;
    padding: 7px 10px;

    -moz-border-radius: 5px;
    border-radius: 5px;

    a {
      font-size: 12px;
```

```
      font-weight: 600;
      color: #fff;
    }
  }
}
```

Em seguida temos o estilo geral do conteúdo do site, por isso fica dentro da `div` com id `content`, como deixamos no layout da aplicação (`app/views/layouts/application.html.erb`). Nesse trecho é interessante observar como podemos fazer referência a um seletor pai usando `&`.

```
#content {
  text-align: left;
  width: $content-width;
  margin: 0 auto;

  h1 { font-size: 1.5em; }

  a, a:visited, a:hover {
    color: #242;
    &:hover { text-decoration: underline; }
  }

  ul, form, p {
    margin: 10px 0;
  }
}
```

Para o formulário, temos o seguinte CSS, sem segredos:

```
form {
  label {
    display: block;
    margin: 5px 0;
    color: #444;
  }

  input[type=text], input[type=password], textarea {
    color: #444;
    font-size: 12px;
    border: 1px solid #ccc;
    padding: 5px;
```

```
    width: 200px;
    outline: 0;

    @include shadow(rgba(0,0,0, 0.1), 0px, 0px, 8px);
    &:focus { border: 1px solid #c9c9c9; }
  }

  textarea {
    width: 400px;
    height: 200px;
  }
}
```

Por fim, temos as classes relacionadas com a informação de erros e `flash`:

```
.field_with_errors {
  display: inline;

  label { color: $error-text-color; }

  input[type=text], input[type=password], textarea {
    border: 1px solid rgba(189,74,72, 0.5);
    @include shadow(rgba(189,74,72, 0.2), 0px, 0px, 8px);

    &:focus { border: 1px solid rgba(189,74,72, 0.6); }
  }
}

.error_message {
  margin-left: 5px;
  display: inline;
  color: $error-text-color;
}

.padded_flash {
  padding: 10px;
  margin: 10px 0;
  font-weight: bold;
  width: 500px;
}

#error_explanation {
```

```
  border: 1px solid $error-text-color;
  color: $error-text-color;
  background-color: #F2DEDE;
  @extend .padded_flash;
}

#notice {
  color: $success-text-color;
  border: 1px solid $success-text-color;
  @extend .padded_flash;
  background-color: #DFF0D8;
}

#alert {
  color: $error-text-color;
  border: 1px solid $error-text-color;
  @extend .padded_flash;
  background-color: #F2DEDE;
}
```

Pronto, esse é todo o estilo que vamos aplicar ao site por enquanto. Salve o arquivo e dê uma navegada. Bem melhor, não? Mas ainda não estamos 100% concluídos. Precisamos corrigir o formulário de erros, e é o que vamos fazer agora.

7.8 Feedback em erros de formulário

Abra o arquivo `app/views/users/_form.html.erb`. Removeremos as tags `<br/>`, pois nossos *labels* estão com `display: block` e simplificaremos a parte superior com a notificação de erros antiga. O problema com essa notificação é que fica difícil para o usuário corrigir facilmente cada campo. Agora, vamos colocar a notificação de erro imediatamente ao lado do campo. Veja o exemplo para o campo "nome completo":

```
<p>
  <%= f.label :full_name %>
  <%= f.text_field :full_name %>
  <% if @user.errors.has_key? :full_name %>
    <div class="error_message">
      <%= @user.errors[:full_name].first %>
    </div>
```

```
  <% end %>
</p>
```

O que fizemos no código anterior, apesar de parecer complicado, é simplesmente verificar se há erro em um atributo específico (neste exemplo, o nome completo). Caso haja erros, vamos criar uma tag `div` com classe `error_message` e mostramos o primeiro erro, para não confundir o usuário com muitas mensagens de erro. O resultado é o seguinte:

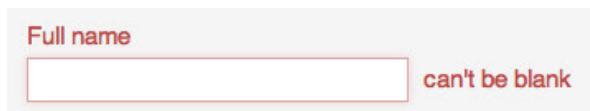



Figura 7.5: Campo com notificação de erro

7.9 Duplicação de lógica na apresentação nunca mais. Use os Helpers

Se você continuou e colocou o código para mostrar a mensagem de erro no formulário para todos os campos, deve ter percebido como ficou poluído e repetitivo. Isso é um alarme! Primeiro que, para cada campo do formulário, repetimos uma lógica (um `if`) e segundo que é bastante trabalhoso. Imagine se quisermos colocar *feedback* de erro também para o formulário de quartos?

Mas não faz sentido nenhum colocar esse tipo de lógica em qualquer uma das camadas que vimos, ou seja, esse código não pertence nem a controles e nem a modelos. É por isso que o Rails disponibiliza um local para você colocar esse tipo de lógica: *view helpers*, ou mais conhecido apenas como *helpers*.

Os *helpers* são lugares adequados para este tipo de lógica, com o objetivo de tornar os templates mais simples e elegantes.

Cuidado com código complexo em helpers!

Lembre-se que código complexo nunca é bom. E pior ainda quando eles estão em *helpers*. Se você estiver escrevendo *helpers* com muitas linhas de código, ou muito `if`, tome isso como um alerta. Você pode estar escrevendo regras de negócio na camada errada, ou talvez você possa estar tentando fazer muita coisa em um lugar só. Tente colocar as regras dentro do modelo ou, se pertinente, crie classes para te ajudar a estruturar os dados de uma forma mais interessante para o template.

Vamos então criar o nosso primeiro *helper*. Os geradores do Rails acabam criando um por controle, e é uma forma bastante interessante de organizá-los. Porém, como vamos usar o *helper* em outros controles, vamos colocar no *helper* geral, o `ApplicationHelper`. Portanto, abra o arquivo `app/helpers/application_helper.rb`:

```
module ApplicationHelper
  def error_tag(model, attribute)
    if model.errors.has_key? attribute
      content_tag :div, model.errors[attribute].first,
        :class => 'error_message'
    end
  end
end
```

Os *helpers* são incluídos em cada template, portanto temos acesso aos modelos via variáveis de instância, porém isso não é uma boa prática. Passamos o modelo como parâmetro, junto com o atributo que queremos verificar por erros. O restante é o mesmo que fizemos quando direto no template. A diferença é que ao invés de gerar o HTML, usamos um *helper* do Rails para criar a tag para nós.

Você pode pensar, por que não retornamos simplesmente o HTML como texto e imprimimos o resultado no template? A razão é que o Rails, por padrão, faz o *escaping* de todo o conteúdo que não é marcado como seguro, evitando injeção de código malicioso. Portanto, é mais seguro e mais simples usar os métodos do próprio Rails.

O resultado visual é o mesmo, mas o template fica muito mais limpo:

```
<p>
  <%= f.label :full_name %>
  <%= f.text_field :full_name %>
  <%= error_tag @user, :full_name %>
</p>
```

Por fim, o template `app/views/users/_form.html.erb` fica assim:

```
<% if @user.errors.any? %>
  <div id="error_explanation">
    Há erros no formulário, por favor verifique.
  </div>
<% end %>

<%= form_for @user do |f| %>
  <p>
    <%= f.label :full_name %>
    <%= f.text_field :full_name %>
    <%= error_tag @user, :full_name %>
  </p>
  <p>
    <%= f.label :location %>
    <%= f.text_field :location %>
    <%= error_tag @user, :location %>
  </p>
  <p>
    <%= f.label :email %>
    <%= f.text_field :email %>
    <%= error_tag @user, :email %>
  </p>
  <p>
    <%= f.label :password %>
    <%= f.password_field :password %>
    <%= error_tag @user, :password %>
  </p>
  <p>
    <%= f.label :password_confirmation %>
    <%= f.password_field :password_confirmation %>
    <%= error_tag @user, :password_confirmation %>
  </p>
  <p>
    <%= f.label :bio %>
```

```
    <%= f.text_area :bio %>
    <%= error_tag @user, :bio %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>
```

Salve o arquivo e envie o formulário, com erros. Ficou legal, né? Como exercício para você leitor, faça o mesmo para o formulário de quartos, que foi gerado pelo *scaffold*. Aproveite e adicione algumas validações no modelo `Room`, pois não há nenhuma, e teste os erros de formulário.

Capítulo 8

Faça sua aplicação falar várias línguas

Lembrar-se que morreremos um dia é a melhor forma que conheço para evitar a armadilha de pensar que temos algo a perder.
– Steve Jobs

8.1 O processo de internacionalização (I18n)

Por enquanto, todo o nosso formulário está em inglês, inclusive os erros. Fizemos dessa forma de propósito, pois vamos usar o sistema de internacionalização (apesar de, em português, a palavra "internacionalização" possuir 19 letras, em inglês ela possui 20, *internationalization*, e portanto é normalmente abreviada por I + 18 letras + n, ou i18n) do próprio Rails.

O sistema de I18n do Rails é bastante simples. Ele funciona com um conjunto de pares chave valor, sendo que uma chave é um identificador de uma mensagem (por

exemplo `users.sign_up.success`) e o valor é o seu texto (por exemplo "Cadastro realizado com sucesso"). Todo o conjunto possui uma chave raiz que é o idioma (`en` para inglês e `pt-BR`, para o português brasileiro).

Meu site só vai ser em português, preciso usar o I18n?

É sempre uma boa ideia deixar seu sistema sempre traduzido com o I18n. A grande vantagem é que os templates ficarão muito mais limpos, mais fáceis de serem trabalhados e revistos por pessoas não técnicas.

Na pasta `config/locales` ficam os arquivos contendo essas traduções, no formato YAML. Se você abrir o arquivo `config/locales/en.yml`, que já vem com o Rails, você observará a seguinte estrutura:

```
en:
  hello: "Hello world"
```

Isso significa que, quando o sistema estiver usando o *locale* "en" (inglês), quando traduzirmos a chave `hello`, o resultado será "Hello world".

Vamos começar colocando as mensagens já traduzidas do Rails, e depois vamos traduzir os nossos atributos.

Os contribuidores do Rails já fizeram a tradução dos erros de validação e diversas outras coisas para português! Portanto o nosso trabalho fica bastante facilitado, basta baixar o arquivo de tradução (vá para http://colcho.net/rails-i18n e clique em "Raw") e salvar como `config/locales/rails.pt-BR.yml`. O nome do arquivo não importa, mas vamos separar nossas mensagens das mensagens do Rails para fins de organização.

Após salvar o arquivo, vamos alterar o idioma padrão do Colcho.net para português brasileiro. Para isso, abra o arquivo `config/application.rb` e procure a seguinte linha:

```
# The default locale is :en and all translations from
#     config/locales/*.rb,yml are auto loaded.
# config.i18n.load_path +=
#     Dir[Rails.root.join('my', 'locales', '*.{rb,yml}').to_s]
# config.i18n.default_locale = :de
```

Descomente e altere o parâmetro para `:'pt-BR'`:

```
# The default locale is :en and all translations from
#     config/locales/*.rb,yml are auto loaded.
# config.i18n.load_path +=
#     Dir[Rails.root.join('my', 'locales', '*.{rb,yml}').to_s]
config.i18n.default_locale = :'pt-BR'
```

Aproveite também e acerte o fuso horário:

```
# Set Time.zone default to the specified zone
#     and make Active Record auto-convert to this zone.
# Run "rake -D time" for a list of tasks
#     for finding time zone names. Default is UTC.
config.time_zone = 'Brasilia'
```

Em seguida, reinicie o servidor. Vá a um dos formulários e tente enviar os dados em branco. *Et voilà*! Erros em português:

Figura 8.1: Erros traduzidos com o sistema de i18n

Certo, agora temos que traduzir os atributos do modelo `User`. Para fazer isso é, basta seguirmos as convenções do Rails. Crie o arquivo `config/locales/pt-BR.yml`:

```
pt-BR:
  activerecord:
    models:
      user: Usuário
    attributes:
```

```
      user:
        bio: Biografia
        email: Email
        full_name: Nome completo
        location: Localização
        password: Senha
        password_confirmation: Confirme sua senha
```

Em seguida, reinicie o servidor do Rails e atualize a página. Agora sim, o formulário faz sentido. Vamos colocar os atributos também para quartos:

```
pt-BR:
  activerecord:
    models:
      room: Quarto
      user: Usuário
    attributes:
      user:
        bio: Biografia
        email: Email
        full_name: Nome completo
        location: Localização
        password: Senha
        password_confirmation: Confirme sua senha
      room:
        description: Descrição
        location: Localização
        title: Título
```

Quando o arquivo YAML já existe, basta atualizar a página, não precisa reiniciar o servidor. Assim, vá ao formulário de novos quartos e verifique se os campos foram traduzidos corretamente.

Figura 8.2: Formulário com atributos traduzidos

8.2 Traduza os templates

Agora que temos a integração de I18n com o `ActiveRecord`, vamos traduzir os templates. Para isso, vamos ter que chamar o mecanismo de I18n quando for necessário. Isso é feito com *helpers* `t` (ou `translate`), para tradução das chaves que já vimos, e `l` (`localize`) para a localização de datas, ou seja, mostrar uma data no formato mais adequado ao idioma corrente.

Vamos traduzir a página de cadastro, que possui apenas um título (`app/views/users/new.html.erb`). Trocamos o texto "Cadastro" pela tradução da chave `users.new.title`:

```
<h1><%= t 'users.new.title' %></h1>
<%= render 'form' %>
<%= link_to t('links.back'), root_path %>
```

Sem criar a tradução no YAML, vá à página e veja o resultado. O Rails acaba imprimindo a última chave numa tentativa de não deixar a página em branco. Porém, o Rails gera um HTML específico para chaves não traduzidas:

```
<h1>
  <span class="translation_missing"
        title="translation missing: pt-BR.users.new.title">
          Title
  </span>
</h1>
...
```

Para corrigir isso, vamos ao YAML de tradução (`config/locale/pt-BR.yml`) e vamos adicionar as chaves de acordo com o que criamos:

```
pt-BR:
  users:
    new:
      title: Cadastro

  links:
    back: Voltar

  activerecord:
  # continua ...
```

Ao recarregar a página, veremos o título e o link traduzidos corretamente.

Dica: estilo para a classe CSS "translation_missing"

No ambiente de desenvolvimento, pode ser interessante colocar um CSS específico para te ajudar a identificar onde faltam elementos para serem traduzidos. Uma maneira de se fazer isso é criar um novo CSS, `app/assets/stylesheets/translation_missing.css.erb`, com o seguinte conteúdo:

```
<% if Rails.env.development? %>
  .translation_missing {
    border: 3px dashed red;
  }
<% end %>
```

Assim, o estilo só será aplicado no ambiente de desenvolvimento. Em ambientes de teste e produção, nada será gerado, sem impacto ao CSS final.

Há uma pequena melhoria que podemos fazer. O Rails automaticamente coloca o nome do controle e da ação no "escopo" das chaves de tradução caso você queira, bastando iniciar a chave com ".". Por exemplo, se mudarmos o título para `.title`, a chave utilizada será `users.new.title`, portanto vamos mudar a chave de tradução:

```
<h1><%= t '.title' %></h1>
<%= render 'form' %>
<%= link_to t('links.back'), root_path %>
```

Aplicando essa mesma lógica na ação *edit* (`app/views/users/edit.html.erb`), teremos:

```
<h1><%= t '.title' %></h1>
<%= render 'form' %>
<%= link_to t('links.back'), @user %>
```

> **Traduções para quartos**
>
> Os templates de quarto podem ser traduzidos da mesma maneira. Aproveite para fazer o mesmo nos outros templates de quarto: `app/views/rooms/new.html.erb` e `app/views/rooms/edit.html.erb`.

A ação *show* será um pouco diferente. Como no título temos o nome do usuário, precisaremos descobrir como passar parâmetros para o sistema de I18n. Isso é feito usando uma notação usando `%{}`. Veja como fica o título dessa página:

```
pt-BR:
  users:
    new:
      title: Cadastro
    edit:
      title: Editar perfil
    show:
      title: "Perfil: %{user_name}"
      edit: 'Editar perfil'
      location: "Localização: %{location}"
      bio: "Bio: %{bio}"

  # ...
```

Veja que o uso de aspas é obrigatório para esse caso. Para usar no template, basta passar um hash mapeando cada atributo a ser interpolado:

```
<h1><%= t '.title', :user_name => @user.full_name %></h1>

<ul>
  <li><%= t '.location', :location => @user.location %></li>
  <li><%= t '.bio', :bio => @user.bio %></li>
</ul>

<%= link_to t('.edit'), edit_user_path(@user) %>
```

Estamos quase prontos, falta pouco!

Vamos à *partial* de formulário (`app/views/users/_form.html.erb`):

```
<% if @user.errors.any? %>
  <div id="error_explanation">
    <%= t 'general.form_error' %>
  </div>
<% end %>
...
```

O layout (app/layouts/application.html.erb):

```
...
      <nav>
        <ul>
          <li><%= link_to t('layout.rooms'), rooms_path %></li>
          <li><%= link_to t('layout.signup'), new_user_path %></li>
        </ul>
      </nav>
...
```

E no YAML (config/locales/pt-BR.yml):

```
# ...
links:
  back: Voltar

layout:
  rooms: Quartos
  signup: Cadastro

general:
  form_error: Há erros no formulário, por favor verifique.

activerecord:
# ...
```

Pronto, todos os templates estão prontos para receberem outros idiomas! No próximo capítulo, iremos trabalhar na última parte da funcionalidade de cadastro: o envio de e-mail de boas vindas e confirmação de conta. Porém, antes de continuar, um pequeno adendo.

8.3 Extra: alterar o idioma do site

Vimos durante este capítulo como usar o mecanismo de internacionalização do Rails e configuramos o idioma padrão como português brasileiro, mas não vimos nenhuma maneira de usar outro idioma a qualquer momento. Vamos deixar o Colcho.net apenas em português, mas é importante para você leitor saber qual é a melhor forma de fazer seu site suportar vários idiomas.

Então, qual é a melhor forma? Sem dúvidas, a melhor maneira de suportar idiomas diferentes no seu site é especificando o idioma via URL. Por exemplo, para o idioma inglês, seria interessante colocarmos a URL `www.colcho.net/en/users/new`.

A principal razão é o *caching*: Como o *caching* é feito por URL, ter páginas diferentes (uma para cada idioma) em uma mesma URL pode tornar o *cache* mais difícil de ser construído e entregue. Quando o idioma faz parte da URL, não há necessidade de alterações no mecanismo de *caching*.

Portanto, vamos ter que adicionar algum mecanismo nas rotas para que contenha o idioma, sem afetar o que já construímos. Isso é possível através do uso de `scope` na construção das rotas:

```
Colchonet::Application.routes.draw do
  scope ":locale" do
    resources :rooms
    resources :users

    resource :user_confirmation, :only => [:show]
  end

  root :to => "home#index"
end
```

A notação `":locale"`, como se fosse um símbolo, significa que todo o conteúdo naquele segmento de caminho na verdade é um parâmetro (ou seja, acessível via o hash `params`). Se você usar os `helpers` de rota do Rails, esse parâmetro será automaticamente incluído sempre que pertinente, portanto não é necessário alterar nenhum outro código.

O que é um segmento de caminho?

Segmento de caminho (*path segments*) é uma parte da URL que é separada por "/" (como se fossem "pastas"). Por exemplo, na imagem a seguir, o caminho (*path*) é `/users/new`, sendo `users` e `new` segmentos distintos:

Figura 8.3: Algumas partes de uma URL

Isso significa que, em toda requisição, o que estiver na URL no segmento `:locale` será passado como parâmetro. Por isso, nos controles conseguimos acesso a essa variável. No `ApplicationController` (`app/controllers/application_controller.rb`), podemos fazer o seguinte:

```
class ApplicationController < ActionController::Base
  protect_from_forgery

  before_filter :set_locale

  def set_locale
    I18n.locale = params[:locale]
  end
end
```

Algumas coisas importantes aconteceram nesse código. Como o `ActiveRecord`, os controles também possuem *callbacks*. No caso de controles, podemos criar filtros através de *class macros*: `before_filter` para métodos executados antes da ação, `after_filter` para depois, ou em volta, via `around_filter`. O filtro mais útil e mais usado é o `before_filter`.

A *class macro* `before_filter` aceita como parâmetro um símbolo representando um nome de método ou um lambda, igual aos *callbacks* do `ActiveRecord`

(isso não é por acaso - o mecanismo de *callbacks* é o mesmo). Como criamos esse filtro no `ApplicationController`, todos os outros controles que herdam do `ApplicationController` (no nosso projeto, todos os controles herdam dela) também terão esse filtro, fazendo com que esse filtro execute em **todas** as ações do site.

Usamos o filtro para configurar o sistema de I18n a usar o idioma que quisermos via o parâmetro da URL. Depois de alterar o arquivo de rotas e o controle, poderá ver que você não consegue mais acessar as URLs normalmente. Colocando o idioma no caminho (http://localhost:3000/pt-BR/users/new), é possível ver a página que traduzimos em português. Se colocarmos `en` no lugar de `pt-BR`, vamos ver a página em inglês, sem as traduções que fizemos apenas ao português:

Figura 8.4: Site em inglês: marcação vermelha onde não temos tradução

Se você tiver interesse, pode copiar o `config/locales/pt-BR.yml` para `config/locales/en.yml` e traduzir o site todo para inglês ou outros idiomas que você quiser.

Constraints em rotas

Um problema do nosso arquivo de rotas atual é que, se um usuário esperto mudar o segmento `en` para algo que ele queira mas que não exista no site, como `jp`, vai

ver um site incompleto ou até mesmo não funcional. Portanto, podemos criar uma *constraint* na rota, ou seja, uma restrição, fazendo com que a rota não seja detectada no caso do idioma não ser suportado.

Podemos usar expressões regulares para criar restrições. Usaremos uma expressão regular que apenas detecta se o `:locale` é `en` ou `pt-BR`:

```
Colchonet::Application.routes.draw do
  scope ":locale", :locale => /en|pt\-BR/ do
    resources :rooms
    resources :users

    resource :user_confirmation, :only => [:show]
  end

  root :to => "home#index"
end
```

Isso fará com que toda vez que a requisição chegar na aplicação, o Rails irá avaliar a expressão regular e vai verificar se o *match* foi feito. Se não for feito, a rota será ignorada. Com essa alteração, se o usuário tentar acessar o site com o `:locale` como `jp`, ele vai receber o erro "404 Não encontrado", pertinente para essa situação.

O esquema das rotas está quase completo. Existem porém um problema complicado: se o usuário tentar acessar a *home*, ou rota raiz do site? O Rails não conseguirá criar as rotas para os quartos na listagem, e o motivo é simples: as rotas para os controles `RoomsController` e `UsersController` só existem se o parâmetro `:locale` existir e, no caso da *home*, ela não existe.

Segmentos opcionais nas rotas

Para solucionar o problema, vamos tornar a presença do `:locale` opcional, tornando o idioma padrão do site como português. Assim, no caso da rota de cadastro, teremos as seguintes possibilidades:

- `/users/new` - cadastro em português;
- `/pt-BR/users/new` - cadastro em português;
- `/en/users/new` - cadastro em inglês;
- `/jp/users/new` - erro "página não encontrada" (`jp`, `es`, ou qualquer outro idioma que não seja `pt-BR` ou `en`)

Para segmentos opcionais, o Rails usa a mesma notação de campos opcionais em expressões regulares:

```
Colchonet::Application.routes.draw do
  scope "(:locale)", :locale => /en|pt\-BR/ do
    resources :rooms
    resources :users

    resource :user_confirmation, :only => [:show]
  end

  root :to => "home#index"
end
```

Ainda falta a rota na *home* que responde a diferentes idiomas. No caso, se você acessar a *home*, ela será tratada como se não houvesse nenhum idioma, indo para o padrão português. Para que a raiz também responda a outros idiomas, temos que adicionar uma nova rota:

```
Colchonet::Application.routes.draw do
  LOCALES = /en|pt\-BR/

  scope "(:locale)", :locale => LOCALES do
    resources :rooms
    resources :users

    resource :user_confirmation, :only => [:show]
  end

  match '/:locale' => 'home#index', :locale => LOCALES
  root :to => "home#index"
end
```

A rota do tipo `match` é o tipo de rota mais simples. Você pode desenhar qualquer rota e ainda usar a notação `":segmento"` para usar segmentos dentro do `params`. Tome cuidado, porém, pois rotas assim podem deixar suas rotas mais complexas e ainda podem ocasionar em bugs chatos de entender quando elas entram em conflito. Adicionamos também a restrição de idiomas, extraindo a expressão regular para uma constante.

Um problema que essa rota opcional gera é que todos os links da sua aplicação teriam que passar a opção `:locale => I18n.locale` para **todos** os *helpers* de rota.

O Rails consegue centralizar essa opção se implementarmos um método chamado `default_url_options`.

Precisamos também atualizar o filtro para que, caso o `:locale` não esteja definido, vamos usar o idioma configurado como padrão no arquivo `config/application.rb`.

Por fim, o `ApplicationController` (`app/controllers/application_controller.rb`) deverá ficar da seguinte forma:

```
class ApplicationController < ActionController::Base
  protect_from_forgery

  before_filter :set_locale

  def set_locale
    I18n.locale = params[:locale] || I18n.default_locale
  end

  def default_url_options
    { :locale => I18n.locale }
  end
end
```

Se você achou complicado, não se assuste. Esse tipo de funcionalidade é uma funcionalidade considerada avançada, com diversas ressalvas e pegadinhas. Contudo, usamos as funcionalidades do Rails para tornar esse processo menos doloroso.

Note também que existem outras maneiras consideráveis de se alterar o idioma, como por exemplo via diferentes domínios, via *headers* HTTP ou até GeoIP, um banco de dados que, através do IP do usuário, determina-se o país e por consequência, seu idioma. Essas opções, apesar de interessantes, não serão abordadas. Se você quiser saber como implementar essas e mais opções, verifique o guia oficial do Rails no assunto: http://guides.rubyonrails.org/i18n.html.

Agora vamos começar a trabalhar no login do usuário!

Capítulo 9

O cadastro do usuário e a confirmação da identidade

"Suor mais sacrifício é igual a sucesso"
– Charles Finley

9.1 Entenda o ActionMailer e use o MailCatcher

O Rails nos dá uma ferramenta de envio de emails chamada `ActionMailer`. Ela funciona como se fosse um controle, ou seja, possui diversos pontos de entrada e por fim resulta em um template que será enviado por email para o usuário.

Para fins de teste, iremos utilizar uma ferramenta **excelente** para enviar e-mails. Ela chama-se `mailcatcher`, uma gem Ruby que disponibiliza dois serviços: um servidor de envio de emails SMTP, que será usado pelo Rails, e um servidor Web, no qual podemos observar os emails que foram enviados, como uma caixa de entrada.

Para instalá-la, basta executar:

```
$ gem install mailcatcher
 ...
Successfully installed mailcatcher-0.5.7.1
8 gems installed
```

Ao executar o comando `mailcatcher`, temos:

```
$ mailcatcher
Starting MailCatcher
==> smtp://127.0.0.1:1025
==> http://127.0.0.1:1080
*** MailCatcher now runs as a daemon by default. Go to the web interface
to quit.
```

Uma vez em execução, vamos voltar ao Rails. Crie o arquivo `app/mailers/signup_mailer.rb` da seguinte forma:

```
class SignupMailer < ActionMailer::Base
  default :from => 'no-reply@colcho.net'

  def confirm_email(user)
    @user = user
    @confirmation_link = root_url # Mudaremos no futuro

    mail({
      :to => user.email,
      :bcc => ['sign ups <signups@colcho.net>'],
      :subject => I18n.t('signup_mailer.confirm_email.subject')
    })
  end
end
```

A classe `SignupMailer` herda de `ActionMailer::Base`, que contém todo o ferramental necessário para envio de emails. Na primeira linha usamos a *class macro* para criar valores *default* para todos os emails a serem enviados.

Em seguida, definimos a nossa "action" para o *mailer*, chamado de `confirm_email`. Associamos duas variáveis de instância, para serem usados no template, da mesma forma que os controles fazem. Por fim, criamos o email, colocando os campos "to:", "bcc:" e o assunto (*subject*) do email. Note que, para termos acesso aos métodos de I18n fora dos templates, precisamos mencionar o módulo `I18n`.

Emails via BCC

É sempre uma boa ideia enviar emails para você mesmo usando o BCC. Dessa forma, você pode ver exatamente o que o usuário recebeu, para tentar ajudá-lo caso haja algum problema, como template quebrado ou campos em branco.

Outro uso importante do BCC é para envio de emails em massa, para evitar que os emails de todos os usuários fiquem à mostra.

9.2 Templates de email, eu preciso deles?

Vamos agora criar os templates para essa "ação". Sim, templates no plural. Podemos criar templates com HTML normalmente, mas é uma boa prática sempre entregar o email com um template em texto puro também, de forma que leitores de emails possam escolher a representação mais adequada. Se criarmos os dois templates da maneira correta, o Rails já faz o trabalho de criar um email que contém as duas representações (chamado de *multipart*) e enviar para o destinatário.

Crie a pasta signup_mailer na pasta app/views e lá vamos criar o template confirm_email.html.erb:

```
<h1><%= t '.title' %></h1>
<p>
  <%= t '.body', :full_name => @user.full_name %>
</p>
<p>
  <%= t '.confirm_link_html',
      :link => link_to(@confirmation_link, @confirmation_link) %>
</p>
<p>
  <%= t '.thanks_html', :link => link_to('Colcho.net', root_url) %>
</p>
```

Agora vamos fazer o template para o email em texto puro. Crie o arquivo app/views/signup_mailer/confirm_email.text.erb com o seguinte conteúdo:

```
<%= t '.title' %>
```

```
<%= t '.body', :full_name => @user.full_name %>
<%= t '.confirm_link_html', :link => @confirmation_link %>

<%= t '.thanks_html', :link => root_url %>
```

Atenção aos links nos mailers

Preste atenção nos links que estamos criando nos templates de *mailer*. Estamos usando o sufixo **_url** e não o **_path**, já que quando o usuário receber o email, ele precisará receber o link completo, e não apenas o caminho. Parece óbvio, mas como sempre usamos **_path** nos templates em geral, é comum cometer essa confusão.

Para essas traduções, adicione as seguintes linhas no YAML do idioma português (`config/locales/pt-BR.yml`), tomando o cuidado para colocar no nível correto, já que a identação importa:

```
pt-BR:
  # ...

  signup_mailer:
    confirm_email:
      subject: 'Colcho.net - Confirme seu email'
      title: 'Seja bem vindo ao Colcho.net!'
      body: |
          Seja bem vindo ao Colcho.net, %{full_name}.
          O Colcho.net é o lugar ideal para você alugar aquele quarto
          sobrando na sua casa e ainda conhecer gente do mundo inteiro.

      confirm_link_html: 'Para você começar a usar o site, acesse o
              link: %{link}'
      thanks_html: 'Obrigado por se cadastrar no %{link}.'

  links:
  # ...
```

Chaves com sufixo _html

Na seção 7.8 falamos de *escaping* de tags HTML e injeção de código malicioso. O sistema de I18n também se preocupa com isso. Então, para evitar que todos os seus links virem texto, é necessário colocar o sufixo `_html` nas chaves de tradução, para que o texto final seja um HTML válido.

Vamos testar! Assumindo que temos um usuário cadastrado no banco de dados, vamos tentar enviar um email para ele:

```
SignupMailer.confirm_email(User.first)
# ArgumentError: Missing host to link to!
#   Please provide the :host parameter,
#   set default_url_options[:host], or set :only_path to true
```

O problema está na hora de gerar links. Pode parecer simples, mas para o servidor Web saber em que endereço ele está não é uma tarefa trivial. Na verdade, ele não tem como saber, dadas as maneiras que um servidor pode ser configurado. Por isso, o que o Rails faz é confiar nos cabeçalhos HTTP (vários deles, dependendo de onde veio a requisição) para montar a URL completa.

O problema é que, frequentemente, os mailers são executados fora do contexto de requisição web, ou seja, em uma tarefa Rake, ou em *jobs* em segundo plano, e, portanto, impossível saber o host do servidor. Dessa forma, o Rails requer que você, manualmente, defina o host para essas situações. Essa configuração pode ser definida por cada ambiente (desenvolvimento, teste e produção). Vamos corrigir para o nosso ambiente de desenvolvimento.

Para isso, abra o arquivo `config/environments/development.rb` e adicione a linha relacionado a `default_url_options`, antes do `end` no final:

```
# ...
  # Expands the lines which load the assets
  config.assets.debug = true

  # Aponta o host para o ambiente de desenvolvimento
  config.action_mailer.default_url_options = {
    :host => "localhost:3000"
  }
end # Não adicione esse end, é contexto apenas
```

Após reiniciar o console, conseguimos enviar o email:

```
SignupMailer.confirm_email(User.first).deliver
# => #<Mail::Message:70211024451560, Multipart: true ... >
```

Objetos criados e email teoricamente enviado, mas isso não significa muita coisa para nós, não é mesmo? Vamos configurar o Rails para que ele possa enviar emails de verdade, usando o serviço de SMTP do `mailcatcher`.

Para isso, abra novamente o `config/environments/development.rb` e coloque as seguintes linhas abaixo da linha que você adicionou anteriormente:

```
# Aponta o host para o ambiente de desenvolvimento
  config.action_mailer.default_url_options = {
    :host => "localhost:3000"
  }

  config.action_mailer.delivery_method = :smtp
  config.action_mailer.smtp_settings = {
    :address => "localhost",
    :port => 1025
  }
end
```

Reinicie o console do Rails e tente novamente:

```
SignupMailer.confirm_email(User.first).deliver
# => #<Mail::Message:70316580535500, Multipart: true, ...>
```

Aponte o seu browser para o endereço do cliente do `mailcatcher` (http://localhost:1080) e veja seus emails. Note que você pode ver também a versão puro texto. Excelente ferramenta, não é mesmo? Agora, precisamos entregar o email no momento do cadastro. Abra o controle `UsersController` (`app/controllers/users_controller.rb`):

```
class UsersController < ApplicationController
  # ...

  def create
    @user = User.new(params[:user])
    if @user.save
      SignupMailer.confirm_email(@user).deliver
```

```
      redirect_to @user, :notice => 'Cadastro criado com sucesso!'
    else
      render :new
    end
  end

  # ...
end
```

Lembre-se de reiniciar o servidor do Rails, caso não o tenha feito após as alterações no arquivo `config/environments/development.rb`. Ao fazer um cadastro, o email será disparado e capturado pelo `mailcatcher`.

Com isso, terminamos o envio de emails. O que nos resta agora é colocar o link de verdade para a confirmação, ao invés de apontar para a página principal.

9.3 Mais emails e a confirmação da conta de usuário

Vamos agora criar a última parte que resta para terminar a funcionalidade de cadastro, que é a confirmação de email. O que devemos fazer é: enviar o email ao usuário e, quando ele clicar em um link único para a conta dele, a conta estará confirmada. Ainda não temos login, porém iremos limitar o acesso do usuário enquanto sua conta não for confirmada. Vamos deixar tudo pronto para que, no próximo capítulo, possamos focar na lógica do login.

A ideia dessa funcionalidade é ter dois novos campos no modelo Usuário: **confirmation_token** e **confirmed_at**. O primeiro campo, `confirmation_token`, vamos gerar no momento do cadastro do usuário e enviar o link contendo este *token* para o email dele. O *token* tem que ser grande e aleatório, de maneira que seja praticamente impossível ser adivinhado e com grande probabilidade de ser único por usuário. O segundo, o `confirmed_at` é o campo que vamos marcar quando o usuário seguir o link recebido no email, tornando o cadastro do usuário totalmente válido.

Para isso, basta criarmos uma nova migração:

```
$ rails generate migration add_confirmation_fields_to_users \
      confirmed_at:datetime confirmation_token:string

invoke active_record
```

```
create db/migrate/
          20120701050227_add_confirmation_fields_to_users.rb
```

Uma coisa interessante é que, quando vamos adicionar campos, se terminarmos o nome da migração com o padrão `add_*_to_*` e em seguida colocar os campos que quisermos, o Rails automaticamente criará o conteúdo da migração:

```ruby
class AddConfirmationFieldsToUsers < ActiveRecord::Migration
  def change
    add_column :users, :confirmed_at, :datetime
    add_column :users, :confirmation_token, :string
  end
end
```

Execute as migrações:

```
$ rake db:migrate
==  AddConfirmationFieldsToUsers: migrating ============
-- add_column(:users, :confirmed_at, :datetime)
   -> 0.0014s
-- add_column(:users, :confirmation_token, :string)
   -> 0.0004s
==  AddConfirmationFieldsToUsers: migrated (0.0020s) ===
```

9.4 Um pouquinho de callbacks para realizar tarefas pontuais

Agora vamos gerar o *token*. Para isso, vamos usar o mecanismo de *callbacks* do `ActiveRecord`. O `ActiveRecord` possui uma série de eventos durante o processo de salvar um objeto. Por exemplo, ao **criar** um objeto, a ordem de chamada é a seguinte:

1) `before_validation` - Antes da validação

2) `validate` - Executa as validações no modelo

3) `after_validation` - Após todas as validações

4) `before_save` - Antes de salvar

5) `before_create` - Antes de criar

6) `create` - Executa a criação do modelo

7) `after_create` - Depois de criar

8) `after_save` - Depois de salvar

9) `after_commit` - Depois de finalizar a transação no banco de dados

O mesmo acontece para atualização de um modelo, a diferença é que, ao invés de usar a palavra-chave `create`, usamos `update`. Em cada um desses eventos é possível plugar código para que possamos executar alguma operação de nosso interesse.

Vamos usar esse mecanismo e plugar um código no evento `before_create`, ou seja, antes da criação do modelo no banco, para preencher o *token* automaticamente.

Cuidado com os callbacks

É muito conveniente usar esses *callbacks*, mas é **muito** importante ter cautela por inúmeras razões. A primeira e mais óbvia é que, se você fizer muitas operações, salvar um modelo torna-se algo imprevisível e demorado. A segunda é que você pode acabar gerando *loops* infinitos e situações complicadas de depurar. Tome bastante cuidado e faça coisas muito simples nesses *callbacks*. Se precisar de lógicas complexas, crie métodos e chame-os quando adequado.

No modelo `User` (`app/models/user.rb`), vamos adicionar o código responsável para gerar o *token*:

```
class User < ActiveRecord::Base
  # Omitindo conteúdo anterior...

  has_secure_password

  before_create :generate_token

  def generate_token
    self.confirmation_token = SecureRandom.urlsafe_base64
  end
end
```

O before_create aceita símbolo com o nome do método a ser chamado ou um lambda. Criamos o método generate_token e nele usamos a biblioteca SecureRandom, que gera números aleatórios o suficiente para nosso uso. Vamos usar também uma forma que seja possível ser passado via URLs e o SafeRandom já possui um método para isso.

Para testar, crie um novo usuário via o browser e depois vá ao console do Rails:

```
User.last.confirmation_token
# => "o8iIOCdUzIXjivrLsfUA8g"
```

Não adicione o confirmation_token ao attr_accessible

Este é um campo que, se permitirmos ser atualizado via formulário, mesmo que de forma indireta, podemos prejudicar a segurança da aplicação, portanto não permita que o campo confirmation_token seja passível de *mass assignment*.

Vamos também criar o método #confirm!, que marca a data e a hora da confirmação e limpa o *token* do usuário, de forma que o link de confirmação só funcione uma única vez:

```
class User < ActiveRecord::Base
  # ...

  def generate_token
    self.confirmation_token = SecureRandom.urlsafe_base64
  end

  def confirm!
    return if confirmed?

    self.confirmed_at = Time.current
    self.confirmation_token = ''
    save!
  end

  def confirmed?
```

```
    confirmed_at.present?
  end
end
```

O método `confirm!` marca o campo `confirmed_at` com a hora corrente, limpa o *token* e salva o modelo. Experimente no console:

```
user = User.last
# => #<User id: 1, full_name: "Vinicius Baggio Fuentes", ...>

user.confirm!
# => true

user.confirmed?
# => true

user.confirmation_token
# => ""
```

Com este código, estamos quase prontos! Agora temos que criar a rota, o controle e corrigir o link no *mailer*.

9.5 Roteamento com restrições

Vamos adicionar uma nova rota no arquivo `config/routes.rb`:

```
Colchonet::Application.routes.draw do
  resources :rooms
  resources :users

  resource :confirmation, :only => [:show]

  root :to => "home#index"
end
```

Note que não estou usando `resources`, mas sim `resource`, no singular. O motivo é que, essencialmente para quem está navegando (ou o cliente de uma API), não existe mais de uma confirmação (ou seja, não faz sentido existir uma ação *index*) no sistema. Isso é chamado de *recurso singleton*. Em seguida, passamos uma opção, que é `:only => [:show]`. Isso significa que só queremos que a ação *show* seja criada, que é a ação que mais se aproxima do que queremos fazer.

Para entender o que essa rota gera, podemos executar o comando `rake routes` na raiz do projeto:

```
$ rake routes
...

confirmation GET  /confirmation(.:format) confirmations#show
```

Vamos criar o controle `ConfirmationsController`. Note que ainda temos que usar o plural no nome. Então criemos o arquivo `app/controllers/confirmations_controller.rb`, com o seguinte conteúdo:

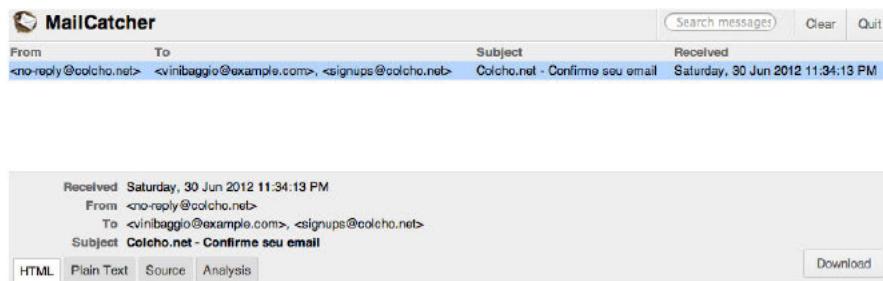
```
class ConfirmationsController < ApplicationController
  def show
    user = User.find_by_confirmation_token(params[:token])

    if user.present?
      user.confirm!
      redirect_to user,
        :notice => I18n.t('users.confirmations.success')
    else
      redirect_to root_path
    end
  end
end
```

9.6 Métodos espertos e os finders dinâmicos

O código desse controle não é complicado e nada de diferente, exceto pela terceira linha:

```
user = User.find_by_confirmation_token(params[:token])
```

Nessa linha estamos usando o que muitas pessoas consideram uma das partes mágicas do Rails, os *dynamic finders*, ou "buscadores" dinâmicos. Para cada atributo do seu modelo, o `ActiveRecord` pode gerar um método que faz uma busca por aquele atributo, sem a necessidade de declararmos esses métodos em lugar algum. Nesse caso, estamos buscando pelo primeiro modelo cujo `confirmation_token` vêm do parâmetro `token`.

Lembrando de adicionar a chave `user.confirmations.success` no YAML de traduções (`config/locales/pt-BR.yml`):

```
pt-BR:
  users:
    confirmations:
      success: Email confirmado com sucesso, obrigado!

  # ...
```

E finalmente, vamos corrigir o link no *mailer* do cadastro (`app/mailers/signup_mailer.rb`):

```
class SignupMailer < ActionMailer::Base
  default :from => 'no-reply@colcho.net'

  def confirm_email(user)
    @user = user
    @confirmation_link = confirmation_url({
      :token => @user.confirmation_token
    })

    mail({
      :to => user.email,
      :bcc => ['sign ups <signups@colcho.net>'],
      :subject => I18n.t('signup_mailer.confirm_email.subject')
    })
  end
end
```

Pronto, terminamos! Ao fazer um cadastro, recebemos um e-mail no `mailcatcher`:

Figura 9.1: Email de confirmação no mailcatcher

Ao seguir o link no email, temos o seguinte resultado:

Figura 9.2: Email confirmado

Pronto, o cadastro está funcionando por completo! Cansou? Vai tomar um café e dar uma volta no quarteirão, que em seguida vamos fazer o login.

9.7 Em resumo

Passamos os últimos capítulos exclusivamente desenvolvendo a funcionalidade de cadastro. Veja o que você aprendeu:

- **Comandos** - Comandos básicos do Rails e do rake, como geradores, visualizar rotas, executar servidor e console;
- **Migrações** - Aprendeu como gerar migrações que criam tabelas, adicionam índices e colunas;
- **Modelos** - Como criar modelos, incluindo validações e *callbacks*;
- **Segurança** - Como evitar injeção de código malicioso via *helpers*, como evitar ataques via *mass assignment*, como encriptar senha dos usuários e o que é *Cross Site Request Forgery*;
- **Emails** - Envio de emails usando *multipart*, enviando HTML e texto puro ao mesmo tempo;
- **Rotas** - Como construir rotas simples;
- **Asset Pipeline** - Aprendeu como um dos componentes mais complexos do Rails funciona e como trabalhar com ele;
- **Controle** - Controle de fluxo de navegação e mensagens *flash*;
- **I18n** - Internacionalização das mensagens da aplicação;
- **Templates e Helpers** - Criação de templates, layouts, *partials*, *view helpers* e deixar seus templates elegantes;

A lista é enorme, parabéns! Vamos agora à funcionalidade de login.

Capítulo 10

Login do usuário

Foco é uma questão de decidir as coisas que você não irá fazer.
– John Carmack, cofundador da idSoftware

Agora é possível com que o usuário se cadastre, confirmando sua conta através do link enviado no email de boas vindas. Porém, ainda não temos nenhum controle de permissões, portanto qualquer visitante pode mudar o perfil de outros usuários, o que não é algo bom!

Vamos então fazer o recurso "sessão", que será criado toda vez que um usuário realizar o login e dura o tempo de navegação do usuário ou quando ele fizer o "logout" (ou seja, destruir o recurso sessão). Criaremos os *templates* e o controle necessário para responder a esse fluxo e vamos usar a autenticação que o método `has_secure_password` nos dá.

Depois, vamos modificar o modelo usuário, para que ele considere a confirmação de conta.

Por fim, vamos modificar o cadastro de quartos para que ele esteja associado ao dono e criar as permissões, de forma que apenas o dono do quarto possa remover

ou atualizar um quarto.

10.1 Trabalhe com a sessão

A partir do momento em que um usuário entra na aplicação através do login e manda novas requisições, precisamos saber quem é o usuário sem perguntá-lo a cada instante. Dessa maneira, precisamos guardar essa informação do usuário logado em algum lugar, que é o que chamamos de **sessão**.

A sessão, apesar de não ser mapeada a uma tabela no banco de dados, pode ser criada ou destruída. Uma vez criada, vamos colocar a informação de login de usuário em seus *cookies*, então, em cada requisição, podemos saber a que usuário pertence. Quando destruirmos a sessão, o que fazemos é remover a informação de login dos *cookies* e as requisições não mais serão tratadas como ações de um usuário específico, o que geralmente é feito quando o usuário realiza seu logout.

Vamos começar criando o modelo `UserSession`. Esse modelo vai possuir as seguintes responsabilidades:

- Tradução de atributos (via I18n);
- Validações em formulário;
- Verificar as credenciais do usuário (email e senha);
- Gravar a sessão do usuário nos *cookies*.

Para tradução de atributos e validações de usuários, vamos usar um componente do Rails chamado `ActiveModel`. Ele é responsável por toda a lógica de *callbacks*, validações e traduções do `ActiveRecord`. Não precisaremos gravar nada em banco de dados, mas ainda assim, podemos ter as vantagens de um modelo que se integra bem com o Rails. O `ActiveModel` possui uma lista grande de componentes úteis, alguns exemplos interessantes são:

- `ActiveModel::Callbacks` - *Callbacks* no ciclo de vida, na forma que vimos com o `ActiveRecord`;
- `ActiveModel::Conversion` - Usado para detectar templates e rotas;
- `ActiveModel::Dirty` - Suporte a atributos "sujos", ou seja, detectar quando um atributo foi alterado e guardar o valor antigo;

- ActiveModel::Naming - Também usado para detectar templates e rotas a partir do modelo (tal como fizemos com o form_for);
- ActiveModel::Translation - Usado para traduzir atributos com o I18n;
- ActiveModel::Validations - Validação de atributos (unicidade, presença, etc.).

Alguns componentes são mais focados na integração com o Rails (como o ActiveModel::AttributeMethods), útil para desenvolvedores que estão fazendo bibliotecas e querem maior integração com o Rails (como por exemplo, suporte ao MongoDB (http://mongodb.org), uma forma diferente de armazenar dados). Na documentação de cada componente existe como você deve usá-lo e os benefícios. Veja o exemplo do componente ActiveModel::Observer: http://api.rubyonrails.org/classes/ActiveModel/Observer.html

Os componentes que nos interessam são os ActiveModel::Naming, para podermos usar os formulários e rotas com o objeto de maneira simples, ActiveModel::Translation para tradução dos atributos (email e senha) e finalmente o ActiveModel::Validations, para mostrar erros de email e senha em branco.

Crie então o arquivo app/models/user_session.rb:

```
class UserSession
  include ActiveModel::Validations
  include ActiveModel::Conversion

  extend ActiveModel::Naming
  extend ActiveModel::Translation

  attr_accessor :email, :password

  validates_presence_of :email, :password

  def persisted?
    false
  end
end
```

O ActiveModel::Conversion exige que implementemos um método chamado #persisted?, que é usado para saber se o modelo tem uma chave primária (para

compor rotas como `/users/1`, por exemplo). Como não vamos gravar no banco de dados, sempre retornamos `false`.

Vamos testar essa nova classe. No console do Rails (ou fazer `reload` caso já esteja com o console aberto):

```
session = UserSession.new
# => #<UserSession:0x007fee1d3b72a8>

session.valid?
# => false

session.errors.full_messages
# => ["Email não pode ficar em branco",
      "Password não pode ficar em branco"]
```

Muito útil, não? Com poucas linhas de código temos um comportamento bastante complexo, graças a modularidade do Rails. Ainda há muito o que se trabalhar nessa classe, porém primeiro vamos amarrar todas as camadas juntas, de forma que possamos testar a parte de autenticação.

> **O que vem por aí no Rails 4: melhorias no ActiveModel**
>
> O `ActiveModel` é relativamente novo, e portanto, mesmo que seja bastante comum criar modelos como o `UserSession` para diversos fins, ainda não é um processo tão simples e elegante como poderia ser. Sabendo disso, os desenvolvedores do Rails já facilitaram esse processo para nós desenvolvedores. No Rails 4, ainda não lançado, vamos poder substituir **tudo** isso para apenas `include ActiveModel::Model`. Enquanto a nova versão não for lançada, porém, é necessário fazer dessa maneira.

10.2 Controles e rotas para o novo recurso

Vamos então criar as rotas do recurso, que haverão apenas as ações `create`, `new` e `destroy`. Para isso, basta criar mais uma entrada no arquivo `config/routes.rb`:

```
Colchonet::Application.routes.draw do
  scope "(:locale)", :locale => /en|pt\-BR/ do
    resources :rooms
```

```
    resources :users

    resource :confirmation, :only => [:show]

    resource :user_sessions, :only => [:create, :new, :destroy]
  end

  root :to => "home#index"
end
```

Em seguida, criamos o controle `UserSessionsController` (`app/controllers/user_sessions_controller.rb`).

```
class UserSessionsController < ApplicationController
  def new
    @session = UserSession.new
  end

  def create
    # Ainda não :-)
  end

  def destroy
    # Ainda não :-)
  end
end
```

Vamos criar o template para a ação `new`. Para isso, precisamos criar a pasta `app/views/user_sessions` e o template `app/views/user_sessions/new.html.erb`:

```
<h1><%= t '.title' %></h1>

<% if @session.errors.has_key? :base %>
  <div id="error_explanation">
    <%= @session.errors[:base].join(', ') %>
  </div>
<% end %>

<%= form_for @session do |f| %>
  <p>
    <%= f.label :email %>
```

```
    <%= f.text_field :email %>
    <%= error_tag @session, :email %>
  </p>
  <p>
    <%= f.label :password %>
    <%= f.password_field :password %>
    <%= error_tag @session, :password %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>

<p><%= link_to t('.sign_up'), new_user_path %></p>
```

Em alguns casos é necessário adicionar erros ao objeto todo e não a um atributo em si. Nesses casos, colocamos as mensagens em `:base`.

Ao direcionar o browser na página de nova sessão de usuário (http://localhost:3000/user_sessions/new), temos o seguinte:

Figura 10.1: Página de login

As bordas vermelhas estão gritando, então vamos criar as traduções que estão faltando no arquivo I18n (`config/locales/pt-BR.yml`).

```
# ...
  show:
    edit: 'Editar perfil'
    title: "Perfil: %{user_name}"
```

```
    location: "Localização: %{location}"
    bio: "Bio: %{bio}"
# Nova parte:
user_sessions:
  new:
    title: 'Login'
    sign_up: 'Cadastre-se'

signup_mailer:
  confirm_email:
#...
```

No fim do arquivo, vamos criar mais duas "seções", contendo as traduções do `ActiveModel` e a tradução específica do botão de login:

```
room:
      description: Descrição
      location: Localização
      title: Título

# Adicione:
activemodel:
  attributes:
    user_session:
      email: Email
      password: Senha
  errors:
    messages:
      invalid_login: 'Usuário ou senha inválidos'

helpers:
  submit:
    user_session:
      create: 'Entrar'
```

Note a porção final do I18n, um trecho especial para os formulários. Você pode criar uma tradução específica para cada modelo como fizemos, bastando criar as chaves necessárias. O padrão que abrange todos os modelos `ActiveRecord` é o seguinte, extraído da tradução padrão do Rails que baixamos (`config/locales/rails.pt-BR.yml`):

```
# Não coloque no seu pt-BR.yml,
# você já tem isso no rails.pt-BR.yml
pt-BR:
  # ...
  helpers:
    submit:
      create: Criar %{model}
      submit: Salvar %{model}
      update: Atualizar %{model}
```

É interessante dar uma olhada no I18n padrão do Rails caso você tenha interesse em customizar uma mensagem para um modelo específico, para não ficar mensagens muitos genéricas e sem personalidade. Brincando com o nome do modelo como escopo de chaves, você pode personalizar a mensagem sem ter que deixar o template poluído.

Agora vamos para a estrela desse capítulo, criando a sessão do usuário!

10.3 Sessões e cookies

Vamos nos lembrar de uma lição muito importante: **não** é interessante colocar a lógica de negócio nos controles. O que seria lógica de negócio? Nesse caso, é tudo a ver com o que não é roteamento, mensagem de erro ou decidir a melhor representação para o recurso.

Dessa forma, o que vamos fazer no controle é apenas tentar criar a sessão via o modelo `UserSession`. Se algum erro acontecer, simplesmente vamos mostrar ao usuário o template com os erros, e se ocorrer sucesso, direcionaremos o usuário para a página principal com uma bela mensagem de sucesso.

Porém, essa ação não será exatamente igual às ações que relacionam-se com o `ActiveRecord`. O controle é a entidade que possui o controle de *cookies*, e portanto teremos que passar ao modelo em qual objeto iremos gravar a sessão do usuário (mas não **como** e **quando**, isso é muito importante!).

Nos controles do Rails, temos duas maneiras de acessar os *cookies* dos usuários, via o método `session` e via o método `cookies`. Ambos os métodos se comportam muito parecido com os já famosos `params` e `flash` - na forma de hashes.

Os *cookies* são simples formas de guardar dados entre requisições HTTP que são passados entre o usuário e o servidor em cada requisição, dando uma sensação de estado. Por exemplo, se tivermos um controle cuja ação execute o código seguinte,

o Rails irá criar uma resposta com o *cookie* informado que, por padrão, irá durar apenas a sessão do usuário, ou seja, ao fechar o browser, o *cookie* será removido pelo browser.

```
def create
  cookies[:pergunta] = 'Biscoito ou bolacha?'
end
```

O cabeçalho da resposta HTTP será algo parecido com o seguinte, note, na penúltima linha, o comando para guardar o *cookie* com o que configuramos:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
X-Ua-Compatible: IE=Edge
Etag: "91c95050746190ff27eb34a00497d91b"
Cache-Control: max-age=0, private, must-revalidate
X-Request-Id: 6194eccc0a2d00f4f472d0a79f286dd4
X-Runtime: 0.020849
Content-Length: 2155
Server: WEBrick/1.3.1 (Ruby/1.9.3/2012-04-20)
Date: Tue, 10 Jul 2012 05:57:59 GMT
Connection: Keep-Alive
Set-Cookie: pergunta=Biscoito+ou+bolacha%3F; path=/
Set-Cookie: _colchonet_session=BAh7.....8857; path=/; HttpOnly
```

Por fim, podemos ver no console do browser (Google Chrome, nesse caso) o resultado do *cookie* guardado:

Figura 10.2: Cookie com a chave 'pergunta'

Note que também temos outra chave de *cookie*, a `_colchonet_session`. Essa é gerenciada automaticamente pelo Rails para controlar o CSRF (Cross-Site Request Forgery, visto na seção 6.1).

O que queremos, no final das contas é armazenar o `ID` do usuário autenticado no *cookie*. Portanto, se o email e a senha estiverem ok e a conta estiver confirmada, salvamos o `ID` do usuário no *cookie* e resgatamos o objeto `User` em cada requisição.

Porém, não é inteligente da nossa parte simplesmente gravar o `ID` do usuário no *cookie* e sentar no sofá esperando a bonança. Isso porquê, ao fazer `cookies[:user_id] = 1`, um usuário mal intencionado pode muito bem pegar o *cookie* local e colocar um outro ID e começar a agir como um outro usuário. Esse ataque é conhecido como *Session Hijacking/Theft* ou sequestro ou roubo de sessão.

O Rails já sabe disso e sabe a melhor maneira de prever ataques como esse. E é justamente essa a grande diferença entre o `cookie` e o `session` - o `session` é encriptado usando uma chave "mágica", de forma que o usuário nunca saberá o conteúdo do *cookie* e, portanto, dificilmente conseguirá forjar uma sessão.

O Rails, para fazer essa encriptação (lembre-se que, diferente das senhas que são apenas uma via, a sessão precisa ser lida no servidor), usa uma chave privada. Essa chave privada fica em `config/initializers/secret_token.rb`:

```
# String looooooooonga....
Colchonet::Application.config.secret_token = 'a49......'
```

É muito importante que você nunca divulgue de forma alguma o `secret_token` de sua aplicação. Deixar em controle de versão não tem problema, mas não deixe isso disponível para seus usuários, pois eles podem ter suas contas roubadas!

Se por algum motivo você suspeitar que o `secret_token` deixou de ser segredo ou que você tem usuários com sessões roubadas, você pode substituir o código atual gerando um novo via:

```
$ rake secret
513b43f26ce....
```

Isso tornarão todas as sessões do site inválidas e os usuários terão que fazer um novo login, mas é melhor garantir a segurança das informações.

Sessões e serialização

É importante lembrar que *cookies*, apesar de não houver limite em sua especificação (RFC 2965), *cookies* possuem uma limitação prática de 4kB, portanto tome cuidado com o que você vai gravar.

Outro problema que pode pegar você de surpresa é a serialização. **Sempre** guarde `IDs` para que objetos sempre estejam em sua versão mais atual quando usados. Há objetos também que não podem ser serializados (por exemplo, possuem referências a IO/arquivos) e podem causar exceções na hora de retornar a resposta ao cliente.

Agora que você já sabe tudo de *cookies* e sessões, vamos à nossa implementação das sessões de usuário. Para relembrar, devemos usar o `session` para gravar o ID do usuário caso ele seja autenticado com sucesso. Dessa forma, vamos deixar o controle `UserSessionsController` (`app/controllers/user_sessions_controller.rb`) da seguinte forma:

```
class UserSessionsController < ApplicationController
  def new
    @session = UserSession.new(session)
  end

  def create
    @session = UserSession.new(session, params[:user_session])
    if @session.authenticate
      # Não esqueça de adicionar a chave no i18n!
      redirect_to root_path, :notice => t('flash.notice.signed_in')
    else
      render :new
    end
  end

  def destroy
  end
end
```

Nesse código, atualizamos a ação `new` para incluir a sessão no `UserSession`. Na ação `create`, não fazemos nada de diferente nos controles que fizemos até agora, a

não ser chamar o método `authenticate`, ao invés do tradicional `save`. Vamos agora à parte mais interessante dessa lógica, o modelo `UserSession`.

Primeiro, precisamos criar o método construtor para aceitar os parâmetros do formulário e a sessão que vem do controle:

```
def initialize(session, attributes={})
  @session = session
  @email = attributes[:email]
  @password = attributes[:password]
end
```

O método `authenticate` é o método que verifica os dados entrados pelo usuário. Se tudo estiver correto, guarda a sessão do usuário, caso contrário, adiciona um erro a ser exibido no formulário.

```
def authenticate
  user = User.authenticate(@email, @password)

  if user.present?
    store(user)
  else
    errors.add(:base, :invalid_login)
    false
  end
end
```

Veja que nesse caso estamos delegando a lógica de verificar se o usuário tem o email e senha válido no modelo `User`. Ou seja, é o modelo usuário que deve saber como a autenticação deve ser feita. É importante centralizar esse tipo de lógica para que, se um dia mudarmos a lógica de como o login é feito, basta alterar em apenas um lugar e isso se reflete em todo lugar que usado. Em poucos parágrafos entraremos em detalhe na implementação desse método.

Usando a API de erros do `ActiveModel` é fácil criar um erro customizado. Nessa situação, o erro não é no atributo `email` e nem no atributo `password`. Por essa razão, adicionamos o erro no `base`, que é o objeto como um todo. Como vimos no template, esse erro irá ser exibido em um lugar diferenciado.

Por fim, temos o método `store`, que grava o usuário na sessão:

```
def store(user)
  @session[:user_id] = user.id
end
```

Por fim, o resultado do modelo `UserSession` (`app/models/user_session.rb`) é:

```
class UserSession
  include ActiveModel::Validations
  include ActiveModel::Conversion

  extend ActiveModel::Translation
  extend ActiveModel::Naming

  attr_accessor :email, :password

  validates_presence_of :email, :password

  def initialize(session, attributes={})
    @session = session
    @email = attributes[:email]
    @password = attributes[:password]
  end

  def authenticate
    user = User.authenticate(@email, @password)

    if user.present?
      store(user)
    else
      errors.add(:base, :invalid_login)
      false
    end
  end

  def store(user)
    @session[:user_id] = user.id
  end

  def persisted?
    false
  end
end
```

Vamos agora ao modelo usuário.

10.4 Consultas no banco de dados

Você se lembra que o modelo usuário (classe `User`) possui um mecanismo de confirmação de contas, não é mesmo? Então, a primeira regra que temos que verificar para saber se um usuário é válido é verificar se o `confirmed_at` está preenchido.

Para fazer buscas dessa forma, vamos entender como funciona o rebuscado mecanismo de busca do `ActiveRecord`, que facilita e **muito** a nossa vida, sem ter que se perder em um oceano de comandos SQL. É importante entender, porém, que o `ActiveRecord` faz o possível, mas não é milagroso, e portanto algumas vezes é necessário escrever SQL e entender a consulta SQL gerada.

Para fazer buscas diretas no banco de dados, o `ActiveRecord` possui uma lista de métodos que lembra muito as cláusulas SQL. Por exemplo, já vimos anteriormente que, para limitar o número de objetos em uma consulta, basta usar o método `.limit`. Para fazer condições de busca, usamos o método `.where`, e assim por diante. Veja a lista de métodos a seguir:

- `where` - Mapeia cláusulas `WHERE`;
- `select` - Especifica o que será retornado no `SELECT`, ao invés de `*`;
- `group` - Mapeia cláusulas `GROUP BY`;
- `order` - Mapeia cláusulas `ORDER BY`;
- `reorder` - Sobrescreve cláusulas de ordem de `default_scope` (veremos o que é `default_scope` ainda nesse capítulo);
- `reverse_order` - Inverte a ordem especificada (crescente ou decrescente);
- `limit` - Mapeia cláusula `LIMIT`;
- `offset` - Mapeia cláusula `OFFSET`;
- `joins` - Usado para *inner joins* ou quaisquer *outer joins*;
- `includes` - Faz *joins* automaticamente com modelos relacionados, veremos relacionamentos no capítulo 11;
- `lock` - Trava atualizações de objetos para atualização;
- `readonly` - Torna os objetos retornados marcados como apenas leitura;

- `from` - Mapeia cláusula `FROM`;
- `having` - Mapeia cláusula `HAVING%`;

Escopos

```
User.limit 1
#  User Load (0.1ms)  SELECT "users".* FROM "users" LIMIT 1
# => [#<User id: 11, full_name: "Vinicius Baggio Fuentes", ... >]

User.where :email => 'vinibaggio@example.com'
# User Load (0.1ms) SELECT "users".* FROM "users"
#       WHERE "users"."email" = 'vinibaggio@example.com'

# => [#<User id: 11, full_name: "Vinicius Baggio Fuentes", ... >]
```

Dica: Comandos SQL no console

Para facilitar o seu aprendizado e também tirar muitas dúvidas, é possível fazer com que o `ActiveRecord` imprima o comando SQL gerado no IRB. Para fazer isso, crie o arquivo `.irbrc` na sua pasta *home* com o seguinte conteúdo:

```
if ENV.include?('RAILS_ENV')
  require 'logger'
  Rails.logger = Logger.new(STDOUT)
end
```

Lembre-se de reiniciar o console do Rails, `reload!` não é suficiente.

Mas isso não é tudo. A parte mais interessante é a composição de consultas. Quando chamamos algum dos métodos de consulta (`.limit` e `.where`, por exemplo), o método retorna um objeto especial chamado `ActiveRecord::Relation`. Esse objeto pode, por sua vez, receber chamadas de métodos de consulta, que irá agregar comandos que já foram chamados anteriormente. Veja o exemplo a seguir:

```
User.where(:email => 'vinibaggio@example.com').limit(2)
#  User Load (0.3ms)  SELECT "users".* FROM "users"
#          WHERE "users"."email" = 'vinibaggio@example.com'
#          LIMIT 2
# => [...]

most_recent = User.order('created_at DESC')

most_recent.limit(1)
# SELECT "users".* FROM "users" ORDER BY created_at DESC LIMIT 1

# Note que a chamada ao .limit anterior não altera
# o objeto most_recent
most_recent.where(:email => 'vinibaggio@example.com')
# SELECT "users".* FROM "users"
#  WHERE "users"."email" = 'vinibaggio@example.com'
#  ORDER BY created_at DESC
```

Essa composição de métodos é o que chamamos de **escopo**. Por exemplo, a variável `most_recent` que criamos é um escopo em `User` que sempre irá retornar os usuários em ordem decrescente pela data de cadastro (`created_at`).

Às vezes é bom entender como o *query planner* (componente que planeja como será a melhor forma de executar a consulta SQL) do banco de dados vai executar uma consulta para que possamos fazer otimizações. Em banco de dados tradicionais como MySQL e PostgreSQL, basta executar o `EXPLAIN` na consulta. Com o `ActiveRecord`, conseguimos fazer isso chamando o método `.explain`:

```
User.where(:email => 'vinibaggio@example.com').
     order('created_at DESC').
     explain
# SELECT "users".* FROM "users"
#   WHERE "users"."email" = 'vinibaggio@example.com'
#   ORDER BY created_at DESC

#  EXPLAIN QUERY PLAN SELECT "users".* FROM "users"
#    WHERE "users"."email" = 'vinibaggio@example.com'
#    ORDER BY created_at DESC

# => "EXPLAIN for: SELECT \"users\".* FROM \"users\"
#       WHERE \"users\".\"email\" = 'vinibaggio@example.com'
#       ORDER BY created_at DESC
```

```
#       0|0|0|SEARCH TABLE users
#            USING INDEX index_users_on_email (email=?) (~10 rows)
#       0|0|0|USE TEMP B-TREE FOR ORDER BY"
```

O `ActiveRecord` também permite criar os famosos *named scopes*, ou escopos nomeados, de modo que você chame `.most_recent`, ao invés de `.order('created DESC')`, tornando as chamadas de métodos bastante legíveis. Imagine que a classe `User` tenha o seguinte código:

```
class User < ActiveRecord::Base
  scope :most_recent, order('created_at DESC')
  scope :from_sampa, where(:location => 'São Paulo')

  # ...
end
```

Encoding

Se você tentar executar esse exemplo e esbarrar com o seguinte problema:

```
SyntaxError: user.rb:3: invalid multibyte char (US-ASCII)
user.rb:3: invalid multibyte char (US-ASCII)
/user.rb:3: syntax error, unexpected $end, expecting ')'
  scope :from_sampa, where(:location => 'São Paulo')
```

Isso deve-se ao fato de que o Ruby não reconheceu esse caractere no código-fonte. Para que o caractere seja reconhecido, coloque o seguinte conteúdo na **primeira** linha do arquivo:

```
# encoding: utf-8
```

Dessa forma, podemos chamar os *named scopes* da mesma maneira que chamamos os outros métodos de busca:

```
User.most_recent.limit(5)
# SELECT "users".* FROM "users"
```

```
#   ORDER BY created_at DESC
#   LIMIT 5

User.most_recent.from_sampa
# SELECT "users".* FROM "users"
#   WHERE "users"."location" = 'São Paulo'
#   ORDER BY created_at DESC
```

Ainda é possível criar *named_scopes* com parâmetros. Para isso, você precisa criar um lambda:

```
class User < ActiveRecord::Base
  scope :most_recent, order('created_at DESC')
  scope :from_sampa, where(:location => 'São Paulo')

  scope :from, ->(location) { where(:location => location) }

  # ...
end

User.from('San Francisco, CA').most_recent
# SELECT "users".* FROM "users"
#   WHERE "users"."location" = 'San Francisco, CA'
#   ORDER BY created_at DESC

# Se condições se repetirem, apenas a última será mantida:
User.from('San Francisco, CA').from_sampa
# SELECT "users".* FROM "users"
#   WHERE "users"."location" = 'São Paulo'
```

Por fim, ainda é possível criar um escopo padrão, ou seja, um escopo que será aplicado mesmo se nenhum for definido, via o `default_scope`. Esse escopo não é sobrescrito e se você definir outros `default_scope`, eles são acumulados.

```
class User < ActiveRecord::Base
  default_scope where('confirmed_at IS NOT NULL')

  # ...
end

User.all
#  SELECT "users".* FROM "users" WHERE (confirmed_at IS NOT NULL)
```

```
User.where(:location => 'San Francisco, CA')
#  SELECT "users".* FROM "users"
#     WHERE "users"."location" = 'San Francisco, CA'
#     AND (confirmed_at IS NOT NULL)
```

Cuidados com o `default_scope`

O `default_scope` tem utilidades muito interessantes, como implementar *soft delete*, ou seja, destruir objetos nada mais é do que marcá-lo como destruído e não ser incluso nas buscas. Portanto, um `default_scope` cuja busca filtre esses objetos é uma ótima ideia.

Porém, tome cuidado. Uma das boas práticas da programação sugere que você deve evitar surpresas desagradáveis, e fazer filtros complexos no `default_scope` é uma surpresa bastante infeliz. A razão disso é que o comportamento torna-se implícito e dificilmente o programador que estiver lendo o código (inclusive você mesmo) lembrará ou saberá da existência de um `default_scope`. Se necessário, peque pelo excesso de clareza ao declarar um escopo nomeado e não o contrário.

Antes de voltarmos ao colcho.net, vamos entrar em um pouco mais de detalhes no `where`, pois ele tem algumas funcionalidades imprescindíveis ao desenvolvedor Rails.

Buscas tradicionais

As buscas mais simples são as buscas que já vimos, para buscar com valores exatos. Para isso, basta passar um hash com a chave sendo a coluna e o valor sendo o valor da busca:

```
User.where(:location => 'San Francisco, CA')
# SELECT "users".* FROM "users"
# WHERE "users"."location" = 'San Francisco, CA'
```

Buscas manuais

Infelizmente não é sempre que o `ActiveRecord` nos ajuda e precisamos fazer uma busca manual. Para isso, basta colocarmos a string com o conteúdo da busca

diretamente. Vimos isso também recentemente:

```
User.where('confirmed_at IS NOT NULL')
# SELECT "users".* FROM "users" WHERE (confirmed_at IS NOT NULL)
```

É bastante importante você notar que, nesse caso, não usamos nenhuma entrada do usuário. Dessa forma, não precisamos nos preocupar com SQL Injection (ou injeção de SQL) e podemos fazer a busca de maneira mais direta, se necessário. Isso é bastante útil para usarmos funcionalidades específicas do banco de dados.

O que é SQL Injection?

SQL Injection, ou injeção de SQL é um ataque a um site de forma que um usuário injete códigos SQL e consiga poder de administrador ou roubar dados importantes. Parece um erro simples, mas ainda é um ataque bastante utilizado e testado. Inclusive, a fatídica queda da Playstation Network® em 2011, que afetou o sistema durante meses e milhões de usuários prejudicados, foi uma única injeção de SQL. Veja o exemplo a seguir:

```
email = 'vinibaggio@example.com'

User.where("email = '#{email}'")
#  SELECT "users".* FROM "users"
#    WHERE (email = 'vinibaggio@example.com')
#
# => [#<User id: 11, full_name: "Vinicius ... >]

email = "' OR 1=1) --"
User.where("email = '#{email}'")
# SELECT "users".* FROM "users"
#   WHERE (email = '' OR 1=1) --')
#
# => [#<User id: 11, ... >]
```

Esse é um exemplo bastante simples mas prova o ponto que, se você não tomar cuidado com os parâmetros, é possível que um usuário mal intencionado ganhe acesso irrestrito ao seu site.

Buscas parametrizadas via Array

A forma mais simples de busca parametrizada é a forma clássica de consultas que evitam SQL Injection. Para esse tipo de parametrização, usamos o símbolo `?` na consulta e o `ActiveRecord` irá sanitizar os parâmetros e substituí-los de acordo. Para isso, passe uma Array cujo primeiro valor é a string SQL e o restante serão os valores substituídos.

```
email = "vinibaggio@example.com"
User.where(["email = ?", email])
# SELECT "users".* FROM "users"
#   WHERE (email = 'vinibaggio@example.com')
# => [#<User :id: 11, ...>]

email = "' OR 1=1) --"
User.where(["email = ?", email])
# SELECT "users".* FROM "users"
#   WHERE (email = ''' OR 1=1) --')
# => []
```

Buscas parametrizadas via hash

Se você tiver muitos parâmetros para substituir, usar o `?` acaba ficando ruim pois fica difícil ler a string SQL e entender o que é cada parâmetro. Por isso, é possível usar chaves e símbolos para substituição de parâmetros:

```
User.where(["location LIKE :location AND email = :email", {
  :location => '%CA%',
  :email => 'vinibaggio@example.com'
}])

# SELECT "users".* FROM "users"
#   WHERE (
#     location LIKE '%CA%'
#     AND email = 'vinibaggio@example.com'
#   )
```

Buscas em listas e intervalos

Com o `where` ainda é possível fazer buscas em listas e intervalos via o comando SQL `IN` ou `BETWEEN`. Para fazer esse tipo de busca, basta usar a busca tradicional, usando hashes e um Array:

```
User.where(:id => [1, 10, 11, 20])
#  SELECT "users".* FROM "users"
#     WHERE "users"."id" IN (1, 10, 11, 20)

User.where(:id => 1..10)
# SELECT "users".* FROM "users"
#   WHERE ("users"."id" BETWEEN 1 AND 10)
```

Essas buscas são extremamente úteis em datas, por exemplo:

```
User.where(:confirmed_at => 1.week.ago..Time.now)
# SELECT "users".* FROM "users"
#   WHERE (
#     "users"."confirmed_at"
#       BETWEEN '2012-07-05 07:37:39.902321'
#       AND '2012-07-12 07:37:39.991792'
#   )
```

10.5 Escopo de usuário confirmado

Agora que temos o conhecimento de como fazer buscas e escopos, temos que implementar um método chamado `.authenticate` no modelo usuário. Ele recebe dois parâmetros, email e senha. Com eles, precisamos verificar, em todos os usuários válidos do sistema (ou seja, usuários que confirmaram seu email), qual possui o email e a senha digitados.

Para isso, primeiro vamos criar um escopo nomeado que retorna todos os usuários que confirmaram sua conta. Adicione o seguinte `scope` no modelo `User` (`app/models/user.rb`):

```
class User
  # ...
  scope :confirmed, where('confirmed_at IS NOT NULL')
  # ...
end
```

Em seguida, vamos criar o método `.authenticate`, que faz a verificação do email e senha. Se o usuário existir e for válido, ele será retornado, caso contrário, a busca retornará `nil`:

```
def self.authenticate(email, password)
  confirmed.
```

```
    find_by_email(email).
    try(:authenticate, password)
end
```

Nesse trecho de código usamos um método do Ruby chamado `#try`. Ele é usado quando não sabemos se um objeto é `nil`, podendo chamar métodos de maneira segura. Dessa forma, se o objeto for `nil`, nada é executado e `nil` é retornado. Se o objeto existir, o método cujo nome é o primeiro parâmetro é executado, com todos os parâmetros em seguida repassados. Veja os exemplos a seguir:

```
string = "oba"        # => "oba"
string.try(:upcase) # => "OBA"
string = nil          # => nil
string.try(:upcase) # => nil
```

A grande utilidade do `#try` é evitar criar `if` e manter a legibilidade do código simples. Portanto no método `.authenticate`, buscamos um único usuário cujo email é o email do parâmetro e que sua conta foi confirmada. Se ele existir, tentamos autenticar a senha. Se a senha for válida, retornamos o usuário, caso contrário, temos `nil`.

Voltemos ao modelo `UserSession` (`app/models/user_session.rb`):

```
def authenticate
  user = User.authenticate(@email, @password)

  if user.present?
    store(user)
  else
    errors.add(:base, :invalid_login)
    false
  end
end
```

Nesse método, que já fizemos, o usuário só será retornado caso o email e senha sejam válidos e o usuário esteja confirmado. Essa parte é muito importante: a classe `UserSession` não precisa saber que existe a lógica de confirmação de usuários, e portanto, delegamos essa lógica apenas ao modelo que deve saber disso.

Com todos os objetos interligados, é possível testar o fluxo completo. Ao digitar as credenciais corretamente, temos o seguinte resultado:

Figura 10.3: Login com credenciais corretas

Agora, quando digitamos as credenciais de forma incorreta:

Figura 10.4: Login com credenciais inválidas

Com o mecanismo de login construído, no próximo capítulo vamos melhorar a estrutura da nossa aplicação de forma a mostrar ao usuário logado, na barra de navegação, links para editar o perfil e fazer logout. Caso o usuário não estiver logado, mostraremos um link para que ele possa fazer o login.

Vamos também fazer o controle de acesso de algumas páginas, como por exemplo, impedir que um usuário edite o perfil de outro usuário, e que um usuário só possa cadastrar um quarto quando estiver logado.

Capítulo 11

Controle de acesso

Formação em Ciência da Computação consegue tornar qualquer pessoa um excelente programador tanto quanto estudar pincéis e pigmentos torna qualquer pessoa um excelente pintor.
– Eric Raymond, autor de 'A catedral e o bazar'

Cadastro e login prontos, agora precisamos focar no controle de acesso. Primeiro, vamos alterar nosso template para exibir uma barra de navegação diferenciada para quando o usuário estiver logado. Atualmente, a barra é assim:

Figura 11.1: Barra de navegação sem informações de sessao

Na versão em que não há usuário logado, deverá ficar assim:

Figura 11.2: Barra de navegação sem login

E por fim, quando o usuário estiver logado, o resultado será:

Figura 11.3: Barra de navegação com perfil do usuário

Em seguida, vamos controlar o fluxo para que um usuário só possa editar o seu próprio perfil. Também vamos corrigir as ações de login e cadastro apenas para quando o usuário **não** estiver logado.

Por fim, vamos forçar que, para um usuário cadastrar um quarto, ele precisa estar logado. Junto a isso, vamos associar o quarto cadastrado ao usuário, aprendendo como criar relacionamento entre modelos `ActiveRecord`. Por fim, vamos também controlar o acesso de edição de quartos, dando a permissão apenas a seus donos, uma lição importante de segurança de dados.

11.1 Helpers de sessão

Antes de alterar a nossa barra, vamos criar métodos para nos auxiliar com a sessão do usuário. Como esse conceito é inerente à aplicação inteira, vamos criar os seguintes métodos no `ApplicationController`:

- `user_signed_in?` - Método que verifica se o usuário possui uma sessão autenticada ou não;
- `current_user` - Retorna o objeto da classe `User` que está na sessão atual;
- `require_authentication` - Filtro que força a autenticação de usuários, redirecionando-o à pagina de login caso não esteja logado, e não fazendo nada caso o usuário esteja com uma sessão válida;

- `require_no_authentication` - Filtro para evitar que usuários cadastrados tentem acessar páginas que só deverão ser acessadas quando o usuário não tiver um login (como páginas de cadastro e de login).

Vamos lá! Para criar essa lógica, vamos ter que melhorar um pouco a classe que nos dá a lógica de login, `UserSession` (`app/models/user_session.rb`):

```
class UserSession
  #...
  def current_user
    User.find(@session[:user_id])
  end

  def user_signed_in?
    @session[:user_id].present?
  end
end
```

Pronto, com esses dois métodos no `UserSession` conseguimos fazer tudo o que precisamos no `ApplicationController`:

```
class ApplicationController < ActionController::Base
  delegate :current_user, :user_signed_in?, :to => :user_session

  # ...

  def user_session
    UserSession.new(session)
  end
end
```

Opa! O que aconteceu? Como o `UserSession` possui toda a lógica que queremos, apenas delegamos as chamadas de método, usando o `delegate` do `ActiveSupport`, ou seja, ao chamar o método `#current_user` no controle, a chamada será repassada para o objeto resultante do método `user_session`. A princípio parece ser muito trabalho e muita abstração desnecessária, mas é importante lembrar das responsabilidades de cada componente e fazer tudo de acordo.

Agora, para os filtros:

```
class ApplicationController < ActionController::Base
  # ...
```

```
  def require_authentication
    unless user_signed_in?
      redirect_to new_user_sessions_path,
        :alert => t('flash.alert.needs_login')
    end
  end

  def require_no_authentication
    redirect_to root_path if user_signed_in?
  end
end
```

Este código não é complicado e é bem legível por conta própria. É importante ressaltar um comportamento de filtros: se o filtro executar uma redireção ou alguma renderização de template (via `redirect_to` ou `render`), a ação e/ou filtros seguintes **não** serão executados. Isso é bastante conveniente para a situação dos filtros `require_authentication` e `require_no_authentication`, mas é importante prestar atenção em filtros que você criar no futuro.

Isso é *quase* tudo que precisamos. Os métodos `current_user` e `user_signed_in?` podem ser usados em controles, mas é importante usá-los também nos templates. Para disponibilizar um método do controle nos templates, usamos a *class macro* `helper_method`. Basta fazer, logo após o `delegate`:

```
class ApplicationController < ActionController::Base
  delegate :current_user, :user_signed_in?, :to => :user_session
  helper_method :current_user, :user_signed_in?

  # ...
end
```

Templates da barra

Agora que temos os métodos de controle de sessão do usuário, já é possível aprimorar nossa barra de navegação. Para isso, vamos ao layout da aplicação (`app/views/layouts/application.html.erb`) e vamos extrair a barra de navegação em uma *partial*:

```
...
<header>
```

```
  <div id="header-wrap">
    <h1><%= link_to "colcho.net", root_path %></h1>
    <% if user_signed_in? %>
      <%= render 'layouts/user_navbar' %>
    <% else %>
      <%= render 'layouts/visitor_navbar' %>
    <% end %>
  </div>
</header>
...
```

Agora vamos criar o template `app/views/layouts/_visitor_navbar.html.erb`. Note que tivemos que especificar o caminho da *partial*. Isso deve-se ao fato de que, como o layout é executado em todas as ações, o caminho de busca de templates fica relativo àquela ação. Especificando o caminho, garantimos que, não importa a ação que estivermos executando, a *partial* sempre será achada.

```
<nav>
  <ul>
    <li><%= link_to t('layout.rooms'), rooms_path %></li>
    <li><%= link_to t('layout.signup'), new_user_path %></li>
    <li><%= link_to t('layout.signin'), new_user_sessions_path %></li>
  </ul>
</nav>
```

O resultado é o seguinte:

Figura 11.4: Barra de navegação sem login

A barra de navegação para usuários logados fica da seguinte forma (`app/views/layouts/_user_navbar.html.erb`):

```
<nav>
  <ul>
    <li><%= link_to t('layout.new_room'), new_room_path %></li>
    <li><%= link_to t('layout.rooms'), rooms_path %></li>
    <li>
```

```
      <%= link_to t('layout.my_profile'),
                  user_path(current_user) %>
    </li>
    <li>
      <%= link_to t('layout.signout'), user_sessions_path,
                    :method => :delete %>
    </li>
  </ul>
</nav>
```

Quando queremos fazer o logout, o que realmente precisamos é apagar a sessão do usuário, via a ação `destroy`. De acordo com a nossa rota, para executar essa ação, precisamos fazer `DELETE /user_sessions`. O problema é que essa ação HTTP não é suportada por todos os browsers, portanto o Rails tem um mecanismo para executar esse tipo de operação:

```
<a href="/pt-BR/user_sessions" data-method="delete"
                                             rel="nofollow">
  Logout
</a>
```

O código anterior possui um atributo `data`, válido no HTML 5. Ele é usado para colocar, em templates HTML, dados que possam ser usados de outra forma. O Rails então usa o `data-method` via Javascript. A forma que isso é feito é através da criação de um formulário (usando `POST`), e incluindo um campo chamado `_method`. Isso tudo é usado para simular o `DELETE`. Bastante trabalhoso para o Rails, para você basta incluir o `:method => :delete`.

Por fim, adicionamos novas chaves na seção "layout" do arquivo de I18n (`config/locales/pt-BR.yml`):

```
layout:
  rooms: Quartos
  new_room: Cadastre seu quarto!
  signup: Cadastro
  signin: Login
  my_profile: Meu perfil
  signout: Logout
```

O resultado é:

Figura 11.5: Barra de navegação com perfil do usuário

Todos os links estarão funcionais e traduzidos, porém ainda não implementamos a ação de destruir a sessão do usuário, e é isso que vamos fazer agora.

Logout

Vamos primeiro à classe `UserSession` (`app/models/user_session.rb`). Para removermos um item da sessão, basta associar `nil` ao item:

```
class UserSession
  # ...
  def destroy
    @session[:user_id] = nil
  end
end
```

Isso é suficiente para que todo o login seja desfeito. Agora precisamos chamar esse método no controle `UserSessionsController` (`app/controllers/user_sessions_controller.rb`):

```
class UserSessionsController < ApplicationController
  # ...

  def destroy
    user_session.destroy
    redirect_to root_path, :notice => t('flash.notice.signed_out')
  end
end
```

O método `user_session` foi criado no `ApplicationController` e vamos usá-lo para tornar o uso desse objeto mais fácil. Basta agora incluir essa mensagem no I18n (`config/locales/pt-BR.yml`) e acabamos a barra!

```
flash:
  notice:
    signed_in: 'Login efetuado com sucesso!'
```

```
    signed_out: 'Logout efetuado com sucesso. Até logo!'
  alert:
    needs_login: 'Para continuar, você precisa estar logado'
```

Figura 11.6: Mensagem de logout após clicar no link

11.2 Não permita edição do perfil alheio

Depois do cadastro e do login, ainda é possível fazer uma coisa **bem** ruim: editar qualquer perfil. Para ver isso funcionando, basta ir ao seu perfil e mudar o ID na rota para outro usuário que você tenha cadastrado.

Para prevenir essa situação, vamos usar um filtro: verificamos se o usuário está logado e se o usuário logado é o mesmo que tentamos editar ou atualizar. Vamos ao controle de usuários (app/controllers/users_controller.rb):

```
class UsersController < ApplicationController
  before_filter :can_change, :only => [:edit, :update]

  # ...

  private

  def can_change
    unless user_signed_in? && current_user == user
      redirect_to user_path(params[:id])
    end
  end

  def user
    @user ||= User.find(params[:id])
```

```
  end
end
```

Nesse código, criamos um filtro chamado `can_change` que se aplica apenas às ações `edit` e `update`. No método `can_change`, redirecionamos o usuário à página do perfil que tentamos atualizar a não ser que o usuário esteja logado e que o perfil a ser atualizado é o dele mesmo.

Usamos também nesse filtro uma expressão idiomática de Ruby chamado *memoization* (ou “memorização”). Nessa expressão, apenas associamos a variável (e por consequência, fazemos a busca no banco de dados) caso ela nunca tenha sido iniciada. Você pode pensar nisso como uma espécia de cache de variável.

Aproveitando que estamos no controle de usuários, vamos forçar que, para a página de cadastro, é necessário não estar logado. Para isso, basta usar o filtro que criamos, `require_no_authentication`:

```
class UsersController < ApplicationController
  before_filter :require_no_authentication, :only => [:new, :create]
  before_filter :can_change, :only => [:edit, :update]

  #...
end
```

Por fim, vamos adicionar o mesmo filtro na ação de login (tanto para o formulário quanto para a ação `create`), no controle `UserSessionsController` (`app/controllers/user_sessions_controller.rb`). Vamos aproveitar e filtrar a ação `destroy` para apenas usuários logados (com o único intuito de evitar ver o `flash` mesmo não tendo feito ação alguma):

```
class UserSessionsController < ApplicationController
  before_filter :require_no_authentication, :only => [:new, :create]
  before_filter :require_authentication, :only => :destroy

  # ...
end
```

Por fim, vamos criar esses filtros para o cadastro e atualização de quartos (aproveitamos também para fazer uma limpeza, tal como colocar I18n, remover os comentários e remover o tratamento para respostas em JSON), no controle `RoomsController` (`app/controllers/rooms_controller.rb`):

```
class RoomsController < ApplicationController
  before_filter :require_authentication,
    :only => [:new, :edit, :create, :update, :destroy]

  def index
    @rooms = Room.all
  end

  def show
    @room = Room.find(params[:id])
  end

  def new
    @room = Room.new
  end

  def edit
    @room = Room.find(params[:id])
  end

  def create
    @room = Room.new(params[:room])

    if @room.save
      redirect_to @room, :notice => t('flash.notice.room_created')
    else
      render action: "new"
    end
  end

  def update
    @room = Room.find(params[:id])

    if @room.update_attributes(params[:room])
      redirect_to @room, :notice => t('flash.notice.room_updated')
    else
      render :action => "edit"
    end
  end

  def destroy
```

```
    @room = Room.find(params[:id])
    @room.destroy

    redirect_to rooms_url
  end
end
```

Por que declarar todas as ações nos filtros?

Para todas as opções de segurança, tenha preferência a ser explícito do que implícito. No caso anterior, poderíamos fazer:

```
before_filter :require_authentication,
  :except => [:index, :show]
```

Porém, se um dia criarmos novas ações, elas automaticamente terão o filtro `require_authentication` aplicado e não necessariamente isso é o que nós queremos. Além disso, ao ser explícitos, é bem mais fácil perceber o que está acontecendo e ajuda a nos lembrar a configurar as permissões corretas.

Com isso, concluímos a parte de filtros, fazendo com que o usuário tenha que estar logado (ou não) em algumas situações. Ainda temos que fazer o controle de permissões, ou seja, não permitir que um usuário não possa editar um quarto que não pertença a ele. Porém, ainda não temos relacionamentos no nosso sistema. A funcionalidade de relacionamentos entre objetos no `ActiveRecord` é uma das melhores coisas dele! Você não está empolgado? Você verá como é fácil.

11.3 Relacionando seus objetos

Uma aplicação Web não pode ser completa sem haver uma modelagem em que objetos se relacionam. Relacionamentos podem ser simples, com um objeto se relacionando diretamente a outro (uma conta de usuário relacionando a sua foto de perfil, por exemplo), também conhecido como relacionamentos um-para-um (ou, em inglês, *one-to-one*), ou ainda como `1:1`.

Para implementar um modelo um-para-um, é muito simples: basta que você tenha um campo na sua tabela para guardar o `id` do outro objeto relacionado. Esse `id` é conhecido como chave estrangeira, ou comumente referenciado pela sua sigla em inglês, *foreign key*.

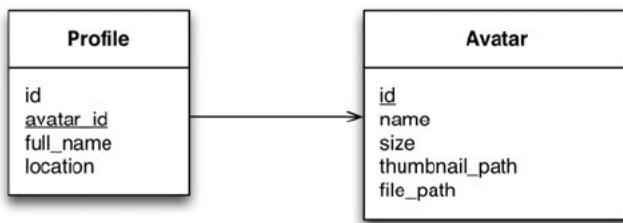



Figura 11.7: Diagrama do relacionamento um-para-um

A modelagem mais popular sem dúvidas é a modelagem um-para-muitos, `1:*` (*one-to-many*, em inglês). Nela um objeto pode possuir nenhum, um, ou até muitos outros objetos relacionados. Um exemplo é o que teremos no próprio colcho.net: um usuário pode ter nenhum quarto, um quarto apenas ou o número que quiser cadastrar, sem restrições.

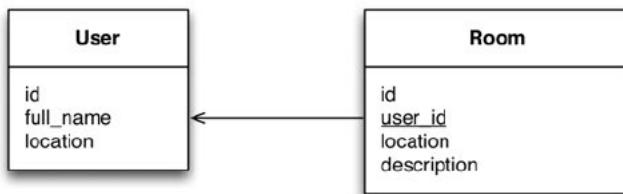



Figura 11.8: Diagrama do relacionamento um-para-muitos

Para a implementação do um-para-muitos basta que mantenhamos a chave estrangeira na tabela cujos objetos pertencem a outro modelo. Assim, para saber quais quartos pertencem a um usuário, por exemplo, basta buscar todos os quartos cujo `user_id` é igual ao `id` do usuário em questão.

Por fim, o tipo de relacionamento mais complexo que temos no Rails é o muitos-

para-muitos, `*:*` (ou *many-to-many*, em inglês). Para essa associação existir, é necessário uma tabela intermediária, chamada tabela de ligação. Imagine uma situação em que um usuário possa participar de vários projetos e que um projeto possua vários membros. É necessário criar uma tabela que contenha duas chaves-estrangeiras, uma apontando ao usuário e outra apontando ao projeto.

É natural que as tabelas de ligação acabem ganhando vida própria como um conceito real dentro do sistema. Ou seja, ao invés de serem uma tabela contendo algumas chaves-estrangeiras, elas são mais que isso. Para o caso do exemplo anterior, é natural chamar a tabela de ligação de "participante".

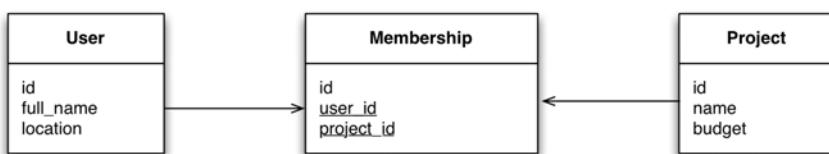



Figura 11.9: Diagrama do relacionamento muitos-para-muitos

O `ActiveRecord` possui facilidades para os três tipos de associações. Vamos aprender na prática como vamos criar uma associação um-para-muitos. Tenho que dizer que as associações um-para-um são muito sem graça e não vamos usar no site. Brincadeiras a parte, associações um-para-um são bem simples de serem criadas e são muito parecidas com o que vamos fazer, que é a associação um-para-muitos.

11.4 Relacione quartos a usuários

O que vamos fazer é criar a associação usuário ao quarto. Um usuário poderá ter nenhum, um, ou vários quartos. Para isso, precisamos adicionar uma chave-estrangeira no modelo `Room` para apontar para `User`. Criamos então uma migração:

```
$ rails g migration add_user_id_to_rooms user_id:integer
   invoke  active_record
   create    db/migrate/20120718060441_add_user_id_to_rooms.rb
```

Como nomeamos a migração no padrão `add_<columns>_to_<table>`, o `ActiveRecord` já gerou a migração com o código de inserção da coluna para nós. Vamos apenas adicionar um índice para facilitar *joins*:

```
class AddUserIdToRooms < ActiveRecord::Migration
  def change
    add_column :rooms, :user_id, :integer
    add_index :rooms, :user_id
  end
end
```

Convenção sobre chaves-estrangeiras

O `ActiveRecord` espera que você nomeie sua chave-estrangeira como `nome_do_relacionamento + _id`, nesse caso, `user_id`. Fazendo dessa maneira, o `ActiveRecord` é capaz de derivar automaticamente o nome da coluna para fazer *join*. Seguir as convenções do Rails às vezes pode ser chato se você está acostumado com outras modelagens, porém a conveniência é bastante grande quando você estiver programando seu sistema. Vamos entender melhor em seguida.

Chaves-estrangeiras são apenas `ids` no banco, portanto não precisamos criar nenhum tipo especial, apenas usaremos inteiros. Note que o `ActiveRecord` não cria nenhum mecanismo de chaves-estrangeiras no banco de dados, portanto se você tiver interesse em fazer isso, deverá executar SQL manualmente, usando o método `execute`, lembrando que isso resultará no acoplamento de suas migrações em um banco de dados específico.

Criando modelos com associações

Nesse caso, estamos criando uma associação após a criação do modelo quarto, portanto estamos criando a chave-estrangeira de um modo mais "baixo nível", adicionando uma coluna de inteiros a uma tabela. Porém, se estivéssemos criando o modelo já com a referência, podemos usar o tipo `references`, que automaticamente teremos a notação de `id` e também o índice. Veja:

```
$ rails g model comment title body:text user:references
```

```ruby
class CreateComments < ActiveRecord::Migration
  def change
    create_table :comments do |t|
      t.string :title
      t.text :body
      t.references :user

      t.timestamps
    end
    add_index :comments, :user_id
  end
end
```

Se você está pensando: "ué, cadê a consistência?", eu concordo com você, deveria ser mais simples, como um `add_references` ou `add_belongs_to`. Mas não é só nós que pensamos assim. No Rails 4, ainda não lançado, vamos ter esses dois métodos.

Após executar `rake db:migrate`, teremos a mais nova coluna no nosso banco de dados. Agora podemos criar as associações nos modelos em si e a parte divertida vai começar!

Vamos editar o modelo `Room` (`app/models/room.rb`) e colocar uma *class macro* para indicar que o modelo quarto pertence a um usuário:

```
class Room < ActiveRecord::Base
  belongs_to :user

  attr_accessible :description, :location, :title

  # ...
end
```

Com a *class macro* `belongs_to`, já é possível associar os objetos na relação quarto -> usuário. O `ActiveRecord` já sabe qual objeto deve criar devido ao nome do relacionamento (`:user`) e também já sabe qual campo usar para buscar o objeto devido (`user_id`):

```
user = User.first
# => #<User id: 11, ... >

room = Room.first
# => #<Room id: 3, ..., user_id: nil>

room.user = user
# => #<User id: 11, ... >

room.save
# => true

room.user
# => #<User id: 11, ...>
```

Com esse relacionamento, já é possível mostrar na página de um quarto, o seu dono. Mas antes de chegar lá, ainda não é possível saber quais quartos um usuário possui. Seguindo todas as convenções do Rails, fazer isso é **muito** simples. Basta adicionar uma única linha no modelo `User` (`app/models/user.rb`):

```
class User < ActiveRecord::Base
  has_many :rooms

  attr_accessible :bio, :email, :full_name, :location, :password,
                  :password_confirmation
  # ...
end
```

A *class macro* has_many faz várias coisas para descobrir o relacionamento. Primeiro, como é um relacionamento um-para-muitos (*has many* significa, literalmente, "tem muitos"), o nome do relacionamento deverá estar no plural e portanto, o modelo é o Room. Dada a natureza do relacionamento, o ActiveRecord sabe também que o modelo room deverá ter um campo para o próprio modelo user e então finalmente, consegue buscar todos os quartos que pertencem a um usuário. Muito conveniente!

```
user = User.first
# => #<User id: 11, ... >
user.rooms
# => [#<Room id: 3, ...>]
```

Note no exemplo anterior como o relacionamento rooms retorna algo *parecido* com um Array. Mas, na verdade, esse objeto nada mais é que um escopo. Isso significa que ainda é possível aplicar qualquer método de busca que vimos na seção "10.4 Consultas no banco de dados", inclusive escopos nomeados!

```
user.rooms.where('title like ?', '%big%')
#  SELECT "rooms".* FROM "rooms"
#     WHERE "rooms"."user_id" = 11 AND (title like '%big%')
#
# => [#<Room id: 3, title: "Big bedroom", ...>]
```

Viu como é fácil criar relacionamento de objetos com Rails? Veremos mais para frente como criar relacionamentos muitos-para-muitos, no capítulo 12, "Avaliação de quartos, relacionamentos muitos para muitos e organização do código".

E o relacionamento um-para-um?

O relacionamento um-para-um é quase igual ao relacionamento um-para-muitos. A diferença é que, ao invés de usar a *class macro* has_many, você usa o has_one (deixando o nome no singular). É importante que o belongs_to **sempre** fique no modelo que possui a chave-estrangeira.

Relacionamentos prontos, vamos dar início ao tratamento de segurança de dados no controle de quartos.

11.5 Limite o acesso usando relacionamentos

Os escopos são ótimos para controlar permissões de acesso e permitir apenas que dados interessantes sejam exibidos. Como os relacionamentos também caracterizam um escopo, vamos usá-lo para limitar o que o usuário pode editar.

Para isso, vamos usar uma técnica interessante: ao invés de fazer a busca (com o `.find`) direto no modelo, sempre que formos criar, atualizar ou deletar um objeto do banco, vamos limitar o escopo de acesso ao usuário logado. Dessa forma, se o usuário logado não tiver permissão para acessar aquele objeto, o usuário irá deparar-se com um erro de Página não encontrada (404).

Usar a associação possui outra vantagem: construir novos objetos usando a associação fará com que o objeto criado já tenha a outra parte do relacionamento ligada corretamente. Veja o exemplo:

```
user = User.first
# => #<User id: 11, ... >
user.rooms.build :title => 'Quarto aconchegante',
                 :description => 'Quarto grande com muita luz natural.'
# => #<Room id: nil,
#         title: "Quarto aconchegante",
#      location: nil,
#   description: "Quarto grande com bastante luz natural.",
#    created_at: nil,
#    updated_at: nil,
#       user_id: 11>

room.user
# => #<User id: 11, ...>
```

Isso é bastante útil para não termos que associar o usuário, neste exemplo, no controle de forma manual, ou ainda pior, no formulário. Vamos aplicar essas ideias no controle `RoomsController`, mas antes vamos fazer uma pausa para uma lição importante: segurança de dados.

O Diaspora (http://joindiaspora.com) é uma alternativa livre ao Facebook (http://facebook.com). Sua proposta é que, ao contrário do Facebook, os dados dos usuários são **realmente** privados e a plataforma possui código-fonte livre para ser investigado. Você pode, por exemplo, instalar o Diaspora em um servidor privado seu e possuir sua própria rede social. Por causa de sua causa nobre, fez bastante barulho nos Estados Unidos no seu lançamento.

Porém, também foi um grande fiasco técnico no lançamento. A plataforma que deveria ser segura e privada possuía grandes falhas de segurança de forma que qualquer usuário poderia ver, criar e editar fotos e outros conteúdos de qualquer usuário. Obviamente os desenvolvedores do Diaspora aprenderam com os erros e já melhoraram a plataforma.

O problema estava nos controles, e temos exatamente a mesma situação ocorrendo no controle do recurso quarto:

```
class RoomsController < ApplicationController
  before_filter :require_authentication,
    :only => [:new, :edit, :create, :update, :destroy]

  # ...
  def update
    @room = Room.find(params[:id])

    respond_to do |format|
      if @room.update_attributes(params[:room])
        format.html { redirect_to @room,
          notice: 'Room was successfully updated.' }
        format.json { head :no_content }
      else
        format.html { render action: "edit" }
        format.json { render json: @room.errors,
          status: :unprocessable_entity }
      end
    end
  end
end
```

O que acontece é que temos o filtro para impedir que um usuário não logado faça atualizações (e outras ações também, como remover) em **qualquer** objeto, mas isso não implica que um usuário não pode atualizar um quarto que não pertence a ele.

Para simular essa falha de segurança é bastante simples. Um usuário pode abrir um formulário de edição de um quarto que pertence a ele, alterar o ID do quarto na URL, e enviar. O código irá buscar pelo quarto, que existe e é valido, e irá atualizar o objeto, de maneira indevida.

A melhor forma de impedir isso é **sempre** usar escopos quando fizermos buscas de objetos. No nosso caso, o usuário atual possui quartos, então vamos usar o

has_many a nosso favor:

```
class RoomsController < ApplicationController
  before_filter :require_authentication,
    :only => [:new, :edit, :create, :update, :destroy]

  # ...
  def update
    @room = current_user.rooms.find(params[:id])

    respond_to do |format|
      if @room.update_attributes(params[:room])
        format.html { redirect_to @room,
          notice: 'Room was successfully updated.' }
        format.json { head :no_content }
      else
        format.html { render action: "edit" }
        format.json { render json: @room.errors,
          status: :unprocessable_entity }
      end
    end
  end
end
```

A única alteração foi a linha 7, ao invés de buscarmos no recurso todo, buscamos pelos quartos que o usuário tem acesso. Dessa forma, mesmo se um usuário mal intencionado alterar o ID do recurso no formulário, a atualização do modelo não irá ocorrer. Isso acontece porque, ao tentar buscar em seus quartos, o objeto não será encontrado, o `ActiveRecord` irá disparar a exceção `ActiveRecord::RecordNotFound` e o controle irá retornar um erro `404 Not found` (não encontrado) para o usuário, o comportamento correto.

Aplicando essa simples mas preciosa lição de segurança, vamos aplicar esse conceito (e aproveitando para fazer umas limpezas no código) no controle `RoomsController` (`app/controllers/rooms_controller.rb`) por completo:

```
class RoomsController < ApplicationController
  def index
    # Exercício pra você! Crie um escopo para ordenar
    # os quartos dos mais recentes aos mais antigos.
    @rooms = Room.most_recent
  end
```

```
  def show
    @room = Room.find(params[:id])
  end

  def new
    @room = current_user.rooms.build
  end

  def edit
    @room = current_user.rooms.find(params[:id])
  end

  def create
    @room = current_user.rooms.build(params[:room])

    if @room.save
      redirect_to @room, :notice => t('flash.notice.room_created')
    else
      render action: "new"
    end
  end

  def update
    @room = current_user.rooms.find(params[:id])

    if @room.update_attributes(params[:room])
      redirect_to @room, :notice => t('flash.notice.room_updated')
    else
      render :action => "edit"
    end
  end

  def destroy
    @room = current_user.rooms.find(params[:id])
    @room.destroy

    redirect_to rooms_url
  end
end
```

> **Quartos já existentes**
>
> Se você já tem algum quarto cadastrado antes dessas alterações, você não vai mais conseguir editá-los ou removê-los. O jeito é destruir todos os quartos (fazendo `Room.destroy_all` no console) ou atualizar todos de modo que você seja o dono (`Room.update_all :user_id => User.first.id`).

Com essa modificações, garantimos a segurança dos dados de forma simples e legível. Muito fácil, não? Agora vamos melhorar os templates para não exibir os links de remoção e edição caso o usuário não seja o dono e vamos aproveitar e dar uma melhorada no visual em geral.

11.6 Exibição e listagem de quartos

Por enquanto, temos a listagem da seguinte forma:

Figura 11.10: Listagem de quartos sem nenhum tratamento

Isso ainda está péssimo. Embora o *scaffold* seja bom para começar e ter uma ideia do que queremos, o resultado dele sempre precisa ser tratado. O nosso objetivo é tornar a listagem um pouco mais elegante:

Figura 11.11: Listagem de quartos com estilo

Antes de criar o template, vamos precisar de um `helper`. Crie o `RoomsHelper` (`app/helpers/rooms_helper`):

```
module RoomsHelper
  def belongs_to_user(room)
    user_signed_in? && room.user == current_user
  end
end
```

Esse código verifica se o usuário está logado e se o quarto pertence a ele. Vamos usar isso para exibir os links de edição e remoção do modelo.

Em seguida, vamos alterar o template da ação `index` (`app/views/rooms/index.html.erb`):

```
<h1><%= t '.title' %></h1>

<% @rooms.each do |room| %>
  <article class="room">
    <h2><%= link_to room.title, room %></h2>
    <span class="created">
```

```
      <%= t '.owner_html',
        :owner => room.user.full_name,
        :when => l(room.created_at, :format => :short) %>
    </span>
    <p>
      <span class="location">
        <%= link_to room.location,
                    "https://maps.google.com/?q=#{room.location}",
                    :target => :blank %>
      </span>
    </p>
    <p><%= room.description %></p>
    <% if belongs_to_user(room) %>
      <ul>
        <li><%= link_to t('.edit'), edit_room_path(room) %></li>
        <li><%= link_to t('.destroy'), room_path(room),
                  :method => :delete, :data =>
                  {:confirm => t('dialogs.destroy')}
            %></li>
      </ul>
    <% end %>
  </article>
<% end %>
```

No template anterior temos uma novidade: o uso do `l`. O `l`, atalho para `localize`, faz parte do sistema de internacionalização, mas com um papel diferente, o de "traduzir" datas e horário, usando o formato adequado para cada idioma. Por exemplo, no Brasil usamos datas no formato dia, mês e ano. Porém, nos Estados Unidos, o mais comum é usar mês, dia e ano. Mas não se preocupe com essas coisas, o I18n trabalha para você, desde que você tenha o arquivo de I18n do idioma.

O `localize` ainda aceita alguns formatos de data, tal como `:short`, `:long` e o padrão (`:default`, no YAML). Você pode ainda criar os formatos que quiser, seguindo o padrão `strftime` (padrão de formatação de hora e data de sistemas POSIX). Veja os formatos que já vem com o Rails:

```
formats:
  default: ! '%A, %d de %B de %Y, %H:%M h'
  long: ! '%A, %d de %B de %Y, %H:%M h'
  short: ! '%d/%m, %H:%M h'
```

Para saber o que é cada símbolo, você pode consultar a documentação do Ruby

no método strftime, ou digitar man strftime, caso você esteja em OS X ou Linux.

Partials de modelos

Uma outra convenção muito útil do Rails são as *partials* de modelos. Veja o exemplo a seguir:

```
<%= render @room %>
```

Se o objeto @room for uma instância do modelo Room, o Rails irá buscar pela *partial* _room.html.erb, e é exatamente o que vamos fazer para deixar o template mais limpo:

```
<% @rooms.each do |room| %>
  <%= render room %>
<% end %>
```

Além disso, o Rails é capaz de renderizar coleções, ou seja, se você passar um Array, por exemplo, o Rails irá renderizar cada elemento da lista. Com ambas as alterações, a listagem de quartos (app/views/rooms/index.html.erb) fica da seguinte maneira:

```
<h1><%= t '.title' %></h1>

<%= render @rooms %>
```

E a *partial* (app/views/rooms/_room.html.erb):

```
<article class="room">
  <h2><%= link_to room.title, room %></h2>
  <span class="created">
    <%= t '.owner_html',
      :owner => room.user.full_name,
      :when => l(room.created_at, :format => :short) %>
  </span>
  <p>
    <span class="location">
      <%= link_to room.location,
                  "https://maps.google.com/?q=#{room.location}",
                  :target => :blank %>
    </span>
  </p>
```

```
  <p><%= room.description %></p>
  <% if belongs_to_user(room) %>
    <ul>
      <li><%= link_to t('.edit'), edit_room_path(room) %></li>
      <li><%= link_to t('.destroy'), room_path(room),
                :method => :delete,
                :data => {:confirm => t('dialogs.destroy')}
            %></li>
    </ul>
  <% end %>
</article>
```

Note que na *partial* fazemos referência ao quarto usando a variável `room`. Isso acontece porque, quando usamos `render` em um objeto, o Rails mapeia o nome do objeto a ser renderizado pelo nome da *partial*. Tome cuidado, pois isso é implícito e fica difícil de descobrir o que está acontecendo em algumas situações. Se você preferir, podemos ser explícito em qual objeto devemos mapear para a *partial*

```
<%= render :partial => 'room', :object => current_user.rooms.first %>
```

O nome do objeto na *partial* ainda será `room`, por causa do nome da *partial*, mas o objeto que estamos renderizando é o `current_user.rooms.first`.

Agora vamos trabalhar no CSS. O que temos que fazer é extrair o *mixin* `shadow` do `default.css.scss` para um arquivo separado para que possamos usar tanto no `default.css.scss` quanto no novo CSS. Para isso, vamos criar um novo CSS para a função de sombra (`app/assets/stylesheets/shadow.css.scss`):

```
@mixin shadow($color, $x, $y, $radius) {
  -moz-box-shadow:    $color $x $y $radius;
  -webkit-box-shadow: $color $x $y $radius;
  box-shadow:         $color $x $y $radius;
}
```

Depois de apagar o *mixin* do `default.css.scss`, temos que incluir a função. Fazemos isso com o uso do `@import`:

```
@import "shadow";

$header-height: 55px;
$content-width: 700px;

...
```

E para finalizar o CSS, vamos criar estilo para quartos (app/assets/stylesheets/room.css.scss):

```
@import "shadow";

.room {
  background-color: white;
  padding: 20px 25px;
  margin-top: 10px;
  @include shadow(#ccc, 0, 3px, 6px);
}

.room h2 {
  display: inline;
  a { font-size: 1.3em; }
}
```

Pronto. A próxima parte é o I18n (config/locales/pt-BR.yml):

```
pt-BR:
  # ...

  dialogs:
    destroy: 'Você tem certeza que quer remover?'

  rooms:
    index:
      title: 'Quartos disponíveis'
    room:
      owner_html: '&mdash; %{owner} (%{when})'
      edit: 'Editar'
      destroy: 'Remover'

  # ...
```

Lembre-se que o _html é necessário para que o I18n retorne entidades HTML (mdash;).

Por fim, a última alteração que precisamos fazer é atualizar o template da ação show (app/views/users/show.html.erb):

```
<%= render @room %>
```

Pronto, terminamos as regras de acesso! No próximo capítulo, vamos aprender como fazer avaliação de quartos, dando uma pontuação de 1 a 5 para um quarto, usando *AJAX* e associações um pouco mais complicadas do que vimos agora.

Capítulo 12

Avaliação de quartos, relacionamentos muitos para muitos e organização do código

Ruby foi construído para tornar os programadores mais felizes.
– Yukihiro “Matz” Matsumoto

A estrutura de dados e templates já está bem completa. O que queremos fazer nesse capítulo é possibilitar a avaliação de quartos, assim novos usuários do colcho.net podem observar a opinião dos outros.

Primeiro, precisamos criar o modelo de avaliação e os relacionamentos entre os modelos `Usuario - User` e `Quarto - Room`, com algumas validações. Com esse relacionamento, vamos em seguida criar as ações de controle para alterar e criar avaliações.

No *front-end*, vamos alterar o template do quarto para incluir as opções de como fazer a avaliação, via AJAX. Para isso funcionar, vamos criar o controle com ações um pouco diferentes, para responder AJAX. O resultado final será o seguinte:

Figura 12.1: Estrelas para avaliação

Por fim, na listagem de todos os quartos, vamos aprender a usar as funções de cálculo (média) do Rails e como executar consultas SQL complexas.

Figura 12.2: Avaliações na listagem com estrelas

12.1 Relacionamentos muitos-para-muitos

Uma avaliação é um modelo que pertence a um quarto e um usuário ao mesmo tempo. Isso significa que para uma avaliação ser única, ela depende de duas chaves estrangeiras: uma para o quarto e outra para o usuário avaliador.

Vamos precisar acessar as avaliações através de um quarto. Por isso, vamos precisar criar o relacionamento no modelo quarto. Isso não será complicado. O outro modelo que precisará ser alterado, como esperado, é o usuário. Precisamos criar o relacionamento de modo que possamos acessar facilmente todos os quartos avaliados pelo usuário.

Para ter esse resultado, vamos:

1) Criar modelo `Review` com chaves estrangeiras `user_id` e `room_id`, além de outros campos;

2) Criar índice para garantir unicidade do par `user_id` e `room_id`, ou seja, um usuário não pode avaliar um mesmo quarto mais de uma vez;

3) Criar validações no novo modelo, como por exemplo, não permitir que o usuário avalie o seu próprio quarto;

4) Criar o relacionamento no modelo quarto;

5) Criar o relacionamento no modelo usuário.

Criando chaves estrangeiras

Vamos usar o gerador para gerar o modelo `Review`, já com os relacionamentos:

```
$ rails g model review user:references room:references points:integer

      invoke  active_record
      create    db/migrate/20120726071529_create_reviews.rb
      create    app/models/review.rb
      invoke    test_unit
      create      test/unit/review_test.rb
      create      test/fixtures/reviews.yml
```

A migração gerada será parecida com:

```
class CreateReviews < ActiveRecord::Migration
  def change
    create_table :reviews do |t|
      t.references :user
      t.references :room
      t.integer :points

      t.timestamps
    end
    add_index :reviews, :user_id
    add_index :reviews, :room_id
  end
end
```

A única coisa que precisamos adicionar nessa migração, é o índice de unicidade no par `[user_id, room_id]`:

```
class CreateReviews < ActiveRecord::Migration
  def change
    create_table :reviews do |t|
      t.references :user
      t.references :room
      t.integer :points

      t.timestamps
    end
    add_index :reviews, :user_id
    add_index :reviews, :room_id

    add_index :reviews, [:user_id, :room_id], :unique => true
  end
end
```

Para efetivar essas alterações no banco de dados, executamos o comando `rake db:migrate`:

```
$ rake db:migrate
==  CreateReviews: migrating ===================
-- create_table(:reviews)
   -> 0.0332s
-- add_index(:reviews, :user_id)
   -> 0.0011s
```

```
-- add_index(:reviews, :room_id)
   -> 0.0007s
-- add_index(:reviews, [:user_id, :room_id], {:unique=>true})
   -> 0.0013s
==  CreateReviews: migrated (0.0366s) ==========
```

Nenhum segredo até então. Nada de complicado também no modelo `Review` (`app/models/review.rb`). O Rails até já criou o modelo com os `belongs_to` necessários, sem contar o `attr_accessible`:

```
class Review < ActiveRecord::Base
  belongs_to :user
  belongs_to :room
  attr_accessible :points
end
```

Vamos usar uma funcionalidade do Rails chamada *counter cache*, ou seja, *cache* de contadores. Como vamos sempre calcular o número de avaliações, o Rails já guarda esse valor pré-calculado em uma coluna do banco de dados automaticamente, desde que usemos a opção `:counter_cache => true` no `belongs_to`:

```
class Review < ActiveRecord::Base
  belongs_to :user
  belongs_to :room, :counter_cache => true
  attr_accessible :points
end
```

Precisamos criar uma nova coluna na tabela de quartos de modo a guardar essa contagem:

```
$ rails g migration add_counter_cache_to_rooms reviews_count:integer
 invoke  active_record
 create    db/migrate/20120812051945_add_counter_cache_to_rooms.rb

$ rake db:migrate
==  AddCounterCacheToRooms: migrating =================
-- add_column(:rooms, :reviews_count, :integer)
   -> 0.0536s
==  AddCounterCacheToRooms: migrated (0.0537s) ========
```

Porém, isso não é suficiente. Precisamos colocar algumas validações no modelo de avaliações:

```
class Review < ActiveRecord::Base
  # Criamos um Array de 5 elementos, ao invés de range.
  POINTS = (1..5).to_a

  belongs_to :user
  belongs_to :room, :counter_cache => true
  attr_accessible :points

  validates_uniqueness_of :user_id, :scope => :room_id
  validates_presence_of :points, :user_id, :room_id
  validates_inclusion_of :points, :in => POINTS
end
```

Essas validações são bem similares com as que já vimos, com exceção da opção `scope` aplicado à validação `uniqueness`. Ela limita o escopo em que a verificação de unicidade ocorre, ou seja, neste exemplo, o `user_id` pode repetir caso o `room_id` seja diferente.

Agora vamos ao modelo `Room` (`app/models/room.rb`). Nele vamos criar o relacionamento um-para-muitos: um quarto possui muitas avaliações.

```
class Room < ActiveRecord::Base
  has_many :reviews
  belongs_to :user

  #...
end
```

Relacionamentos de ponta-a-ponta

Em relacionamentos muitos-para-muitos, é comum precisarmos acessar o modelo da outra ponta da tabela de ligação, ou seja, no Colcho.net, pode ser interessante um usuário saber todos os quartos que ele tem algum voto. A ideia é juntar todos os registros de avaliação que o usuário possui e, através desses registros, buscar os quartos.

O ActiveRecord possui uma maneira de te ajudar com este problema. Uma vez definido o relacionamento has_many :reviews, podemos criar o seguinte relacionamento:

```
class Room < ActiveRecord::Base
  # É necessário definir o has_many primeiro!
  has_many :reviews
  has_many :reviewed_rooms, :through => :reviews, :source => :room

  # ...
end
```

Esse exemplo cria o relacionamento reviewed_rooms que, através de todas as avaliações que um usuário tem (especificado pela opção :through), irá buscar, nesse modelo de ligação, o relacionamento :room, retornando todos os quartos que satisfazem estes relacionamentos.

Por fim, vamos criar o relacionamento no modelo User (app/models/user.rb), para que possamos ter acesso às avaliações daquele usuário:

```
class User < ActiveRecord::Base
  has_many :rooms
  has_many :reviews

  # ...
end
```

Pronto! Todos os relacionamentos foram criados. Antes de continuar, veja o exemplo a seguir:

```
# Lembre-se do sandbox para evitar a perda de dados!
# Para usar o sandbox, basta fazer "rails c --sandbox"

room = Room.last
# => #<Room id: 5, ... >

review = room.reviews.build :points => 3
review.user = User.last
review.save
# => #<Review id: 1, ... >

Review.all
# => [#<Review id: 1, ... >]

room.destroy
# => #<Room id: 5, >

r = Review.first
# => #<Review id: 1, >

r.room
# => nil
```

Diferença entre `#destroy` e `#delete`

O `ActiveRecord` possui dois métodos para destruir objetos no banco de dados: o `#destroy` e o `#delete`. É **muito** importante lembrar que eles possuem comportamentos diferentes.

O `#destroy` é o método que normalmente deve ser usado. Ele executa todos os *callbacks* e deleta o objeto no banco de dados. O `#delete`, por sua vez, apenas executa o `DELETE` no banco de dados.

O que acontece é que removemos o objeto `room`, porém ainda temos referências inválidas no banco de dados. Para arrumar isso, poderíamos criar um *callback* no momento que um objeto está sendo destruído (`after_delete`) e remover os objetos referenciados. Essa solução funciona, porém, o `ActiveRecord` facilita a nossa vida.

12.2 Removendo objetos sem deixar rastros

No momento que vamos destruir um objeto, é possível destruir também objetos relacionados. Para isso, adicionamos uma opção nas associações que queremos remover quando o objeto for destruído, chamada `dependent`. Seu comportamento é muito parecido com a opção `CASCADE` do SQL:

- `destroy` - Executa `#destroy` em todos os objetos associados, executando os *callbacks* de cada um;
- `delete_all` - Deleta os objetos associados via SQL apenas, sem execução de *callbacks*;
- `nullify` - Apenas marca as chaves estrangeiras dos objetos relacionados com `NULL`;
- `restrict` - Impede a remoção do objeto se houver objetos relacionados.

Nesse caso, queremos usar a opção `:destroy`, para que os *callbacks* sejam de fato chamados, precisamos destruir todas as avaliações associadas ao usuário quando ele é destruído. Para isso, basta colocar uma opção no `has_many`. Veja como deve ficar o modelo `User` (`app/models/user.rb`):

```
class User < ActiveRecord::Base
  # Aproveite a oportunidade para atualizar o outro
  # relacionamento:
  has_many :rooms, :dependent => :destroy
  has_many :reviews, :dependent => :destroy

  # ...
end
```

E, por fim, no modelo `Room` (`app/models/room.rb`):

```
class Room < ActiveRecord::Base
  has_many :reviews, :dependent => :destroy
  belongs_to :user

  # ...
end
```

Cuidado com o `:dependent => :destroy`

Apesar da grande conveniência, o `:dependent => :destroy` pode ser perigoso. Como o `ActiveRecord` tem que instanciar cada objeto e chamar o método `#destroy`, que por sua vez, pode ter seus relacionamentos, que vai instanciar todos os objetos, e... Deu pra entender onde isso vai parar, né?

Além de lento, esse procedimento pode ser perigoso. Se o seu banco de dados tiver muitos registros, o Rails irá instanciar um objeto `ActiveRecord` para cada, resultando em muita alocação de memória e muitas interrupções do *garbage collector* para limpar objetos não usados.

Em situações assim, você pode pensar em uma das seguintes soluções:

- Marcar os objetos a serem removidos por um serviço que executa de tempos em tempos;
- Usar `dependent => :delete` e usar chaves estrangeiras com `on delete cascade`, no próprio banco, fazendo com que o banco de dados fique responsável pela limpeza dos dados.

12.3 Criando avaliações com pitadas de AJAX

Pronto, agora que temos o modelo de avaliação, vamos implementar o *front-end*, usar uma nova fonte e incluir o JavaScript e chamadas AJAX.

Para isso, primeiro vamos criar o ponto de entrada para o recurso `Review`. Ele fará parte do recurso `Room`, ou seja, a URL será montada da seguinte forma: `/rooms/:room_id/reviews`. Isso dará ao nosso projeto uma noção de que um quarto possui uma ou diversas avaliações, que é o que estamos procurando.

Criaremos a rota da seguinte forma (`config/routes.rb`):

```
Colchonet::Application.routes.draw do
  #...

  resources :rooms do
```

```
    resources :reviews, :only => [:create, :update]
  end

  # ...
end
```

Ao aninhar o recurso `:reviews` no recurso `:rooms` na rota, alcançamos o aninhamento também na rota. Podemos ver o resultado executando `rake routes`:

```
room_reviews POST (/:locale)/rooms/:room_id/reviews(.:format)
                  reviews#create
                  {:locale=>/en|pt\-BR/}

room_review PUT   (/:locale)/rooms/:room_id/reviews/:id(.:format)
                  rooms/reviews#update
                  {:locale=>/en|pt\-BR/}
```

O Rails irá buscar o controle como `ReviewsController` e, como de costume, ficaria em `app/controllers/reviews_controller.rb`. Porém, depois de alguns meses ou anos de projetos, deixar todos os controles nessa pasta fica uma bagunça sem tamanho. Imagine uma pasta com mais de 100 controles, cada um em uma rota diferente...

Por essa razão, organizaremos controles que possuem aninhamento em módulos. As principais vantagens dessa prática são: extrair comportamento comum, como buscar o elemento a qual o recurso pertence (por exemplo, buscar o objeto `room` no controle de avaliações) ou filtros e a organização dos arquivos.

Portanto, vamos customizar a nossa rota de forma que o Rails busque o controle na pasta `app/controllers/rooms/` e a classe do controle seja `Rooms::ReviewsController`:

```
Colchonet::Application.routes.draw do
  #...

  resources :rooms do
    resources :reviews, :only => [:create, :update], :module => :rooms
  end

  # ...
end
```

Agora vamos criar o controle `Rooms::ReviewsController` (`app/controllers/rooms/reviews_controller.rb`):

```
class Rooms::ReviewsController < ApplicationController
  before_filter :require_authentication

  def create
    review = room.reviews.
      find_or_initialize_by_user_id(current_user.id)

    review.update_attributes!(params[:review])

    head :ok
  end

  def update
    create
  end

  private

  def room
    @room ||= Room.find(params[:room_id])
  end
end
```

Usamos nesse controle um método muito útil do `ActiveRecord`: o *dynamic finder* `.find_or_initialize_by_...`. Em modelos ou associações, o `.find_or_initialize_by_...` irá fazer uma busca pelos atributos mencionados no nome do método. Se não encontrar, um novo objeto será instanciado (mas não salvo no banco de dados) com os atributos passados. No nosso exemplo, faremos a busca por `user_id` e, se não encontrado, o `ActiveRecord` irá criar uma nova instância de `Review` já com o `user_id` marcado como `current_user.id`.

Método similar: `find_or_create_by_...`

O `ActiveRecord` ainda possui outro método, chamado `find_or_create_by_...`. O funcionamento dele é bem parecido com o `%find_or_initialize_by_...`. A diferença é que, ao invés de apenas instanciar o objeto, o `ActiveRecord` cria uma entrada no banco de dados.

Como vamos responder apenas a requisições AJAX, não precisamos renderizar nenhum conteúdo. Portanto, apenas respondemos com o código HTTP `201 Created` quando há sucesso. Nesse caso, como o único *input* do usuário é um valor pré-selecionado, não devemos encontrar erros. Se por um acaso encontrarmos, há algum problema em nosso projeto e portanto usamos o método *bang* no `#update_attributes!`, de forma a disparar exceções e ficar mais fácil de vermos que há algo errado.

Lembramos que só existe uma única avaliação de um usuário a um quarto. Dessa forma, o comportamento do `update` é o mesmo do `create`, só que sempre iremos encontrar o objeto no `find_or_initialize_...`. O restante é o mesmo e, portanto, vamos apenas delegar um método ao outro.

Falha silenciosa vs. falha barulhenta

Quando desenvolvemos aplicações, temos a tendência de tentar tratar ou silenciar erros, evitando que usuários sejam presenciados com uma página de erro.

Porém, é importante salientar que, quando fazemos isso, fica mais difícil descobrir que algo está errado e tentar caçar algum *bug* misterioso. Quando algo inesperado acontece, é melhor ver um *stack trace* do que tentar descobrir, fazendo *debug* por horas para descobrir porque uma variável está `nil`, por exemplo.

Dica: Reuso de código em controles

Não precisamos criar abstrações no nosso exemplo. Porém, na necessidade de compartilhar código entre vários controles em um mesmo módulo, podemos criar uma classe chamada `Rooms::BaseController` e nela colocar filtros e outros métodos interessantes:

```
class Rooms::BaseController < ApplicationController
  before_filter :require_authentication
  private

  def room
    @room ||= Room.find(params[:id])
  end
end
```

Em seguida, basta herdar desse controle e o comportamento será compartilhado:

```
class Rooms::ReviewsController < Rooms::BaseController
  def create
    review = room.reviews.
      find_or_initialize_by_user_id(current_user.id)

    # ...
  end
end
```

Por fim, vamos acertar o controle `RoomsController` (`app/controllers/rooms_controller.rb`) para construir um objeto de avaliação a ser usado no template de quarto:

```
class RoomsController < ApplicationController
  # ...

  def show
```

```
    @room = Room.find(params[:id])

    if user_signed_in?
      @user_review = @room.reviews.
        find_or_initialize_by_user_id(current_user.id)
    end
  end

  #...
end
```

Controles prontos, vamos aos *templates*. Vamos colocar as tradicionais estrelas de avaliação, mas a priori vamos focar na funcionalidade. Vamos usar *radio buttons* para que o usuário escolha a pontuação, de 1 a 5:

Figura 12.3: Avaliação de quartos com radio buttons

Para fazer isso, vamos criar um formulário na *partial* de quartos (app/views/rooms/_room.html.erb):

```
<article class="room">
  <h2><%= link_to room.title, room %></h2>

  <%= render :partial => 'review', :object => @user_review %>

  <span class="created">
    <%= t '.owner_html',
      :owner => room.user.full_name,
      :when => l(room.created_at, :format => :short) %>
  </span>
```

```
  ...
</article>
```

Na *partial* `review` (`app/views/rooms/_review.html.erb`), teremos:

```
<section class="review">
  <% if user_signed_in? %>
    <%= form_for [review.room, review] do |f| %>
      <% Review::POINTS.each do |point| %>
        <%= f.radio_button :points, point  %>
        <%= f.label :points, point, :value => point %>
      <% end %>

      <%= f.submit %>
    <% end %>
  <% else %>
    <span class="login_required">
      <%= t('.login_to_review') %>
    </span>
  <% end %>
</section>
```

A primeira diferença que você vai notar é a construção da rota para o `form_for`. Como o recurso "avaliação" é aninhado ao recurso "quarto", é necessário identificar a qual quarto pertence a nova avaliação que criamos no controle de quartos. Usando a notação de Array, dizemos ao Rails todas as dependências da rota.

Note que isso causa um problema com a listagem de quartos (ação `index`). Vamos resolver esse problema ainda neste capítulo. Por agora, vamos focar na exibição de quartos.

Rotas com *namespaces*

É possível criar rotas com *namespaces*, ou seja, um nome que na verdade não representa um recurso, mas que divide a aplicação em "módulos". Por exemplo, uma área de administração (ou "admin") pode ser um *namespace*. Para declará-los nas rotas, basta fazer:

```
namespace :admin do
  resources :products
end
```

Para você identificar *namespaces* nos formulários, você deve também usar a notação de Array:

```
<%= form_for [:admin, @product] do |f| %>
 ...
```

Quando criamos formulários em HTML, os campos de *label* possuem um atributo chamado `for`, que deve possuir o `name` ou `id` do elemento à que este *label* se associa. Fazendo isso, clicar no texto do *label* irá selecionar o campo de texto, se associado com um campo de texto, selecionar um elemento de um grupo de *radio buttons* e assim por diante.

Para tornar *radio buttons* em um mesmo grupo (ou seja, selecionando um irá desmarcar o outro), é necessário usar um mesmo `name`. Isso quebra com a forma de que o `label` funciona, ou seja, todos os *labels* de um mesmo grupo, se usando os *helpers* do Rails, iriam apontar para um mesmo botão, deixando de funcionar da maneira esperada.

É aí que entra a opção `:value` do *helper* `label`. Você deve usar essa opção tanto para *labels* em elementos do tipo *radio button* ou *check boxes*. O *helper* irá construir o atributo `for` apontando para o elemento correto.

Nesse template, usamos uma nova entrada no I18n (`config/locales/pt-BR.yml`):

```
pt-BR:
  # ...
```

```
  rooms:
    # ...
    review:
      login_to_review: 'Faça o login para avaliar quartos'
```

Por fim, adicione as seguintes regras CSS para os botões alinharem da forma correta e estilizar o texto de login (app/assets/stylesheets/room.css.scss):

```
// Usamos o #content para aumentar a especificidade do seletor,
// ou seja, tornar essa regra mais importante que outras.
#content .review form {
  margin: 0;
}

#content .review {
  margin: 0;
  padding: 0;
  float: right;
  border: none;
}

.review label {
  display: inline;
}

.review .login_required {
  font-size: 0.8em;
  color: #666;
  font-variant: small-caps;
}
```

Figura 12.4: Texto de login necessário

E traduzimos o modelo `Review`:

```
pt-BR:
  # ...
  activerecord:
    models:
      room: Quarto
      user: Usuário
      review: Avaliação
```

12.4 Diga adeus a regras complexas de apresentação: use presenters

Tem algo que muito me incomoda nesse código, para ser sincero. Temos uma regra de *template* repetida tanto no `RoomsController` quanto no *template*.

No `RoomsController`:

```
if user_signed_in?
  @user_review = @room.reviews.
    find_or_initialize_by_user_id(current_user.id)
end
```

E no *template*:

```
<% if user_signed_in? %>
  <%= form_for [review.room, review] do |f| %>
  <%# ... %>
```

```
<% else %>
  <%# ... %>
<% end %>
```

Para resolver esse problema, vamos criar uma classe que vamos usar tanto nos *templates* quanto no controle.

Os *presenters*, ou apresentadores, são responsáveis por fazer essa ligação de uma maneira descomplicada e resolvendo o problema de *templates* ou controles complexos.

Vamos criar então o nosso `RoomPresenter`. Crie o arquivo e a pasta `app/presenters/room_presenter.rb`.

A ideia dessa classe é a seguinte: podemos passar um quarto a qual a avaliação irá pertencer e o contexto que o *presenter* se aplica: pode ser tanto um template quanto o próprio controle. Precisamos deste contexto para saber se o usuário está logado, por exemplo. O restante serve apenas para tornar o template mais elegante.

```
class RoomPresenter
  delegate :user, :created_at, :description, :location, :title,
    :to => :@room

  def initialize(room, context, show_form=true)
    @context = context
    @room = room
    @show_form = show_form
  end

  def can_review?
    @context.user_signed_in?
  end

  def show_form?
    @show_form
  end

  def review
    @review ||= @room.reviews.
      find_or_initialize_by_user_id(@context.current_user.id)
  end

  def review_route
```

```
    [@room, review]
  end

  def route
    @room
  end

  def review_points
    Review::POINTS
  end

  # render @room resulta na renderização da partial 'room'
  def to_partial_path
    'room'
  end
end
```

As ações `show` e `index` do controle `RoomsController` (`app/controllers/rooms_controller`) ficam da seguinte forma:

```
class RoomsController < ApplicationController
  # ...

  def index
    @rooms = Room.most_recent.map do |room|
      # Não exibiremos o formulário na listagem
      RoomPresenter.new(room, self, false)
    end
  end

  def show
    room_model = Room.find(params[:id])
    @room = RoomPresenter.new(room_model, self)
  end
end
```

A *partial* `room` (`app/views/rooms/_room.html.erb`), por sua vez, fica:

```
<article class="room">
  <h2><%= link_to room.title, room.route %></h2>

  <%= render :partial => 'review', :locals => {:room => room} %>
```

```
  <span class="created">

  <%# ... $>
  <% if belongs_to_user(room) %>
    <ul>
      <li><%= link_to t('.edit'), edit_room_path(room.route) %></li>
      <li><%= link_to t('.destroy'), room_path(room.route),
        :method => :delete,
        :data => {:confirm => t('dialogs.destroy')} %>
      </li>
    </ul>
  <% end %>
</article>
```

Por fim, vamos extrair o formulário de review (app/views/rooms/_review.html.erb), em uma nova *partial*, review_form (app/views/rooms/_review_form.html.erb).

O resultado é que a partial app/views/rooms/_review.html.erb ficará como:

```
<section class="review">
  <% if room.show_form? %>
    <%= render :partial => 'review_form', :locals => {:room => room} %>
  <% end %>
</section>
```

E a partial app/views/rooms/_review_form.html.erb ficará da seguinte maneira:

```
<% if room.can_review? %>
  <%= form_for room.review_route do |f| %>
    <% room.review_points.each do |point| %>
      <%= f.radio_button :points, point  %>
      <%= f.label :points, point, :value => point %>
    <% end %>

    <%= f.submit %>
  <% end %>
<% else %>
  <span class="login_required">
    <%= t('.login_to_review') %>
```

```
  </span>
<% end %>
```

12.5 jQuery e Rails: fazer requisições AJAX ficou muito fácil

O código que temos atualmente cria avaliações, porém clicar no botão 'Criar Avaliação' irá nos exibir uma página em branco. A ação foi projetada para que façamos a requisição via AJAX, e é isso que vamos fazer agora.

Você deve estar pensando: "bom, vamos primeiro instalar nosso framework Javascript preferido (*aham* jQuery) e usá-lo para facilitar nossa vida". Certíssimo! Atualmente é difícil o site que use JavaScript e não possua o jQuery, para o bem ou para o mal.

O Rails, como um framework que quer te ajudar a colocar o seu site no ar, já instala o jQuery (http://jquery.com/) em sua aplicação. Melhor que isso: o Rails ainda integra o jQuery em muito dos *helpers*, incluindo formulários via AJAX.

Portanto, para fazer nosso formulário usar AJAX e enviar as requisições assincronamente, basta alterar uma linha.

Altere o `form_for` na *partial* do formulário de *review* (`app/views/rooms/_review_form.html.erb`) para incluir a opção `:remote => true`:

```
<% if room.can_review? %>
  <%= form_for room.review_route, :remote => true do |f| %>
    <%# ... %>
  <% end %>
<% else %>
  <%# ... %>
<% end %>
```

Pronto! Ao usar o `:remote => true`, o Rails irá criar um atributo `data` chamado `data-remote`. A partir daí, um script chamado "Unobtrusive scripting adapter", ou adaptador para script não-intrusivo, irá observar o evento `submit` do formulário, capturá-lo e enviar todos os dados via AJAX, ao invés da forma tradicional.

Esse adaptador está intimamente ligado ao jQuery, mas existem outras implementações dele, fazendo essa ligação a outros frameworks, como o prototype.js (http://prototypejs.org/).

Se você ainda desejar usar outro framework de sua preferência, não é difícil criar o seu próprio, basta remover a entrada `gem 'jquery-rails'` do `Gemfile`, que é quem habilita o jQuery no projeto.

O adaptador também disponibiliza para nós alguns eventos para que possamos registrar *callbacks* e fazer alguma customização de comportamento. Vamos usar esses eventos para travar o botão do formulário durante o envio da requisição (`ajax:beforeSend`) e sinalizar sucesso (`ajax:success`) ou erro (`ajax:error`) ao usuário. A lista completa de eventos é:

- `ajax:before` - Antes de preparar a requisição AJAX (ou seja, antes de capturar os campos do formulário e preparar as opções de envio da requisição);
- `ajax:beforeSend` - Chamado antes de enviar a requisição, porém com tudo já pronto para envio;
- `ajax:success` - Depois do término da requisição e a resposta foi bem sucedida;
- `ajax:error` - Depois do término da requisição e a resposta foi mal sucedida;
- `ajax:complete` - Chamado depois do término da requisição, não importando o resultado.

Desenvolvendo JavaScript manualmente

Se você preferir, é possível não usar o adaptador do Rails e fazer tudo "na mão". O que temos, por fim, é a própria biblioteca jQuery, então podemos usá-la da forma que quisermos.

Por fim, se você quiser, pode usar CoffeeScript, bastando adicionar a extensão `.coffee` e escrever o código equivalente.

Crie o arquivo `app/assets/javascripts/room.js`:

```
$(function() {
  var $review = $('.review');

  $review.bind('ajax:beforeSend', function() {
    $(this).find('input').attr('disabled', true);
```

```
  });

  $review.bind('ajax:error', function() {
    replaceButton(this, 'icon-remove', '#B94A48');
  });

  $review.bind('ajax:success', function() {
    replaceButton(this, 'icon-ok', '#468847');
  });

  function replaceButton(container, icon_class, color) {
    $(container).find('input:submit').
        replaceWith($('<i/>').
                       addClass(icon_class).
                       css('color', color));
  };
});
```

Nesse código, registramos *callbacks* para três eventos: `ajax:beforeSend` (e **não** `ajax:before`), `ajax:error` e `ajax:success`. No evento `ajax:beforeSend`, desativamos todos os *inputs*, pois eles não serão mais operacionais (não podemos votar mais de uma vez). Se tudo der certo, substituímos o botão por um ícone de sucesso ("v") e se der errado, por um "x".

12.6 Média de avaliações usando agregações

Estamos quase terminando a funcionalidade de avaliações! A última coisa que vamos fazer antes de algumas mudanças visuais é exibir a nota média de avaliações do quarto.

Para isso, vamos criar um método no modelo de avaliações para retornar o valor da média de pontos, usando uma função de cálculo do `ActiveRecord`. As funções de cálculo são:

- `average` - média;
- `minimum` - mínimo;
- `maximum` - máximo;
- `count` - contagem;

- sum - soma.

Nesses métodos, você deverá passar o nome do campo cujos valores serão calculados. Poderá passar também algumas opções interessantes. Veja alguns exemplos:

```
Review.average('points')
#  SELECT AVG("reviews"."points") AS avg_id FROM "reviews"
# => #<BigDecimal:7f8b34bd8ed8,'0.21E1',18(45)>

Review.average('points').to_f
# => 2.1

# Funciona com escopos!
Room.first.reviews.average('points').to_f
# SELECT AVG("reviews"."points") AS avg_id
#   FROM "reviews" WHERE "reviews"."room_id" = 4
#
# => 2.25

Review.average(:points, :group => :room)
#  SELECT AVG("reviews"."points") AS average_points,
#        room_id AS room_id FROM "reviews"  GROUP BY room_id
#
# => {#<Room id: 4, ...>=>#<BigDecimal:7f8b349ab318,'0.225E1',18(45)>,
#     #<Room id: 5, ...>=>#<BigDecimal:7f8b349aa7b0,'0.2E1',9(45)>}

Review.maximum(:points)
# SELECT MAX("reviews"."points") AS max_id FROM "reviews"
# => 5

Review.minimum(:points)
# SELECT MIN("reviews"."points") AS min_id FROM "reviews"
# => 1

Review.count
# SELECT COUNT(*) FROM "reviews"
# => 10
```

Uma grande vantagem desses métodos de cálculo é que, como você pôde observar, eles são todos feitos via SQL e portanto podem ser beneficiados pelos índices do banco de dados e não há o custo de alocar um objeto na memória por cada registro.

Usando o método `.average` do `ActiveRecord`, vamos calcular o número de estrelas a partir de uma coleção de avaliações. Usando o método `#round` de números, podemos arrendondar as estrelas para um número inteiro. Juntando esses dois métodos, vamos criar um método de classe no modelo `Review` (`app/models/review.rb`): eles também são aplicados a escopos!

```
class Review < ActiveRecord::Base
  # ...

  def self.stars
    (average(:points) || 0).round
  end
end
```

Veja exemplos de uso:

```
Room.first.reviews.stars
# => 2

Review.stars
# => 2
```

Vamos então usar esse método em nosso *presenter* para quartos (`app/presenters/room_presenter.rb`), criando os métodos `#stars` e `total_reviews`:

```
class RoomPresenter
  # ...

  def review_points
    Review::POINTS
  end

  def stars
    @room.reviews.stars
  end

  def total_reviews
    @room.reviews.size
  end
```

```
  # ...
end
```

Dica: métodos de contagem

Uma vantagem de escopos acaba se tornando uma desvantagem: eles se comportam e se parecem muito como Arrays. Arrays possuem três métodos que contam o número de objetos: `#length`, `#count` e `#size`. Todos eles funcionam da mesma maneira.

Porém, isso não se reflete em escopos e modelos `ActiveRecord`. Os três métodos existem, mas com comportamentos diferentes:

- `#length` - É o mesmo do Array, portanto faz com que o `ActiveRecord` busque todos os objetos no banco, instancie-os e depois faz a contagem;
- `#count` - Conta quantos objetos existem no banco de dados, fazendo uma consulta SQL (vimos agora pouco: faz parte dos métodos de cálculo) ou usando o *counter cache*;
- `#size` - Chama a contagem pelo método `#length` caso os objetos tenham sido carregados, caso contrário, conta via o método `#count`.

Dessa forma, sempre que for fazer contagem de objetos, prefira usar o `#size`!

Vamos mostrar então no *template* de avaliações (`app/views/rooms/_review.html.erb`) esses números:

```
<section class="review">
  <% if room.show_form? %>
    <%= render :partial => 'review_form', :locals => {:room => room} %>
  <% else %>
    <%= t '.stats', :average => room.stars,
                    :max => room.review_points.max,
                    :count => room.total_reviews %>
```

```
  <% end %>
</section>
```

E no arquivo de I18n (`config/locales/pt-BR.yml`), usamos uma funcionalidade para plurais:

```
pt-BR:
  # ...
  rooms:
    #...

    review:
      login_to_review: 'Faça o login para avaliar quartos'
      stats:
        zero: 'Não há avaliações'
        one: '%{average}/%{max} (1 avaliação)'
        other: '%{average}/%{max} (%{count} avaliações)'
```

O que acontece é que, quando usamos a chave `:count` na tradução, podemos criar mensagens diferenciadas para cada valor de `:count`: *zero* para zero, *one* para um e *other* para o restante, assim não precisamos nos preocupar em fazer regras para cada valor.

Figura 12.5: Avaliações com estatísticas básicas

12.7 Aplicações modernas usam fontes modernas

Por muito tempo, designers sofriam na criação de páginas Web. Limitados a não mais que 10 fontes, eles tinham que fazer milagres para tornar um site elegante e não cair em *clichés*. Porém, isso mudou completamente desde que os *browsers* passaram a aceitar fontes embutidas via a diretiva `@font-face`. Com ela, é possível que um site use uma fonte não instalada no sistema do usuário.

Além disso, o Google criou o serviço *Google Webfonts* (www.google.com/webfonts/), serviço gratuito e que possui diversas fontes de alta qualidade e muito fácil de usar. É exatamente isso que estamos usando no Colcho.net para usar a fonte "Pacifico".

Recentemente, as *webfonts* tem sido usadas para outro propósito. Com a criação de dispositivos com altíssima densidade de *pixels* (o "Novo iPad" e os MacBooks com "Retina Display"), os browsers acabam escalonando as imagens para que elas fiquem no tamanho configurado. Isso resulta em defeitos e degradação da qualidade das imagens em um site. Veja o exemplo a seguir:

Figura 12.6: TV5.org: Site não preparado para alta densidade

Veja a diferença entre a qualidade da fonte, que é um elemento facilmente escalonável e as imagens. Quando elas são ampliadas, muitos defeitos visuais ocorrem. Compare com o exemplo do site CSS Tricks (http://www.css-tricks.com), que é otimizado para "Retina Display":

Figura 12.7: CSS-Tricks.com: Otimizado para alta densidade

Alcançar esse resultado não é simples, é necessário aplicar vários truques com CSS e repetir imagens de forma a ter uma versão com pouca e outra com alta densidade de pixels, ou através do uso de imagens SVG (em vetor). Outra solução é transformar seus ícones e imagens bastante utilizadas em fontes. Como as fontes são facilmente escalonáveis (pois são vetores, e não mapa de pixels) e simples de serem criadas, muitos designers e desenvolvedores de *front-end* estão optando por essa solução.

Existe uma fonte *open source* e disponível para uso comercial chamada "Font Awesome" (http://fortawesome.github.com/Font-Awesome/). Além de possuir grande qualidade, ela possui vários pictogramas úteis e é de graça.

Portanto, antes de continuar com o desenvolvimento do site, vamos instalá-la no Colcho.net. Não é trivial como simplesmente copiar e colar, pois temos que adaptá-la ao uso do Assets Pipeline, mas não será difícil. Baixe o pacote no site da fonte, descompacte-o e copie os arquivos de dentro da pasta `fonts/` para a pasta do projeto, em `vendor/assets/fonts`. Copie também o CSS `css/font-awesome.css` para a pasta `vendor/assets/stylesheets`, renomeie-o para `font-awesome.css.scss` e altere a diretiva `@font-face`:

```
@font-face {
  font-family: "FontAwesome";
  src: url(font-path('fontawesome-webfont.eot'));
  src: url(font-path('fontawesome-webfont.eot') + '?#iefix')
    format('eot'),
       url(font-path('fontawesome-webfont.woff')) format('woff'),
       url(font-path('fontawesome-webfont.ttf')) format('truetype'),
       url(font-path('fontawesome-webfont.svg') + '#FontAwesome')
    format('svg');
  font-weight: normal;
  font-style: normal;
}

// O restante é o mesmo...
```

Temos que fazer essa alteração para que o Assets Pipeline gere a rota correta para a fonte, através do `font-path`. Como o `font-path` é uma diretiva SCSS e é pré-processada para gerar o CSS final, renomeamos o arquivo.

A pasta `vendor/assets/fonts` não está no caminho de pesquisa de assets do Assets Pipeline. Para adicionar, basta adicionar a seguinte linha do `config/application.rb`:

```
module Colchonet
  class Application < Rails::Application
    # ...
    config.assets.paths << Rails.root.join("vendor", "assets", "fonts")
  end
end
```

Por fim, adicione a `'font-awesome'` no manifesto CSS (`app/assets/stylesheets/application.css`):

```
/*
 * ...
 *= require_self
 *= require_tree .
 *= require 'font-awesome'
 */
```

Pronto, temos a Font Awesome instalada e pronta para uso!

Como instalar o Font Awesome sem o Assets Pipeline?

Se você não quiser usar o Assets Pipeline para as fontes, basta copiar as fontes para `public/fonts`, copiar o CSS para `public/stylesheets` e acertar a url para `/fonts/fontawesome-webfont.eot` e o mesmo para as outras entradas. Não há necessidade de alterar o `config/application.rb`.

12.8 Eu vejo estrelas - usando CSS e JavaScript para melhorar as avaliações

Com *back-end* e o *front-end*, vamos colocar a "Font Awesome" para uso: vamos usar estrelas para mostrar as pontuações de 1 a 5 na listagem. Para isso, vamos mudar um pouco o *template*, o CSS e o I18n.

Comecemos pelo template de avaliações (`app/views/room/_review.html.erb`):

```
<section class="review">
  <% if room.show_form? %>
    <%= render :partial => 'review_form', :locals => {:room => room} %>
  <% else %>
    <% room.stars.times do %>
      <span class="star filled_star"></span>
    <% end %>

    <% (room.review_points.max - room.stars).times do %>
      <span class="star empty_star"></span>
    <% end %>

    <%= t '.stats', :count => room.total_reviews %>
  <% end %>
</section>
```

Primeiro, criamos o número de estrelas preenchidas (`filled_star`) e depois criamos o número de estrelas não preenchidas (subtraindo o número de estrelas do quarto do total de 5). O resultado é o seguinte:

```
<section class="review">
  <span class="star filled_star"></span>
```

```
  <span class="star filled_star"></span>
  <span class="star filled_star"></span>

  <span class="star empty_star"></span>
  <span class="star empty_star"></span>

  (27 avaliações)
</section>
```

Atualizamos também o I18n (`config/locales/pt-BR.yml`) para alterar a mensagem:

```
pt-BR:
  rooms:
    review:
      login_to_review: 'Faça o login para avaliar quartos'
      stats:
        one: '(1 avaliação)'
        other: '(%{count} avaliações)'
```

E, por fim, o CSS (`app/assets/stylesheets/room.css.scss`):

```
.review .star {
  font-family: "FontAwesome";
  font-size: 18px;
}

.review .filled_star {
  color: gold;
  text-shadow: 1px 1px #999;
  &:before { content: "\f005"; }
}

.review .empty_star {
  color: #aaa;
  &:before { content: "\f006"; }
}
```

Usamos o `:before` para adicionar um caractere especial da "Font Awesome" antes das estrelas.

Pronto! Depois dessas alterações, temos a listagem com estrelas:

Figura 12.8: Avaliações na listagem com estrelas

Agora, para finalizar, vamos tornar a votação um pouco mais interessante, vamos usar as estrelas de modo que o usuário possa votar apenas no clique. Para isso, vamos alterar o CSS de forma que o cursor do mouse indique ao usuário que ele pode interagir com a estrela. Vamos colocar as cores e estilos.

Para isso, primeiro altere o CSS do formulário (`app/assets/stylesheets/room.css.scss`):

```
#content .review {
  margin: 0;
  padding: 0;
  float: right;
  border: none;
  /* Adicione a seguinte linha: */
  position: relative;
}
```

Em seguida, adicione as regras para posicionar os símbolos de OK e erro em um lugar de forma que as estrelas não saiam do lugar quando o formulário for enviado.

```
.review .icon-ok,
.review .icon-remove {
  position: absolute;
  right: 0;
}
```

As próximas 2 regras são para estilizar a estrela de acordo com o novo HTML do formulário, que veremos em seguida.

```
.review label i {
  font-size: 18px;
  text-shadow: 1px 1px #999;
  cursor:pointer;
  color: #ccc;
}

.review label.toggled i {
  color: gold;
}
```

A última regra é para sumir com os botões *radio* e o "enviar".

```
.review form > input {
  display:none;
}
```

O novo template do formulário (`app/views/rooms/_review_form.html.erb`) deve ficar assim:

```
<% if room.can_review? %>
  <%= form_for room.review_route, :remote => true do |f| %>
    <% room.review_points.each do |point| %>
      <%= f.radio_button :points, point  %>
      <%= f.label :points, :value => point do %>
        <i class="icon-star"></i>
      <% end %>
    <% end %>

    <%= f.submit %>
  <% end %>
<% else %>
  <span class="login_required">
```

```
    <%= t('.login_to_review') %>
  </span>
<% end %>
```

A diferença está na maneira que construímos o *label*. Ao invés de colocarmos o valor, vamos simplesmente desenhar uma estrela, de acordo com o "Font-Awesome".

Para encerrar essa funcionalidade, adicionamos o código JavaScript para alterar a cor das estrelas quando o usuário passar o mouse sobre elas, e com isso mostrar ao usuário que ele está de fato alterando a sua avaliação.

Para isso, adicione o seguinte código JavaScript no arquivo `app/assets/javascripts/rooms.js`:

```
$(function() {
  // ...

  function highlightStars(elem) {
    elem.parent().children('label').removeClass('toggled');
    elem.addClass('toggled').prevAll('label').addClass('toggled');
  }

  highlightStars($('.review input:checked + label'));

  var $stars = $('.review input:enabled ~ label');

  $stars.on('mouseenter', function() {
    highlightStars($(this));
  });

  $stars.on('mouseleave', function() {
    highlightStars($('.review input:checked + label'));
  });

  $('.review input').on('change', function() {
    $stars.off('mouseenter').off('mouseleave').off('click');
    $(this).parent('form').submit();
  });
});
```

A primeira função, `highlightStars`, é a responsável por adicionar e remover o destaque das estrelas. Baseado no elemento passado, primeiro remove-se o destaque

de todas as estrelas (classe CSS `toggled`) e em seguida adiciona-se a mesma classe apenas ao elemento em destaque e os anteriores.

Usando essa função, ativamos as estrelas previamente selecionadas pelo usuário. Isso é importante para mostrar ao usuário que a ação dele teve efeito. Este seletor executa no momento que a página é carregada.

A primeira parte, `input:checked` irá retornar todos os *inputs* que estão marcados (válido somente para *check boxes* e *radio buttons*). Usando o +, retornamos apenas o primeiro objeto **imediatamente** ao redor deste *input*. Isso significa que o seletor irá retornar o *label* imediatamente ao lado de um *input* que está selecionado. Veja a seguir o resultado:

Figura 12.9: Formulário de avaliação enviado

```
var $stars = $('.review input:enabled ~ label');
```

Na linha anterior, selecionamos todos os *labels* (as estrelas) em *inputs* que estão habilitados. O comportamento é parecido com o seletor `input:checked + label`, porém dessa vez procuramos **todos** que estão habilitados. O objetivo é que só iremos dar destaque quando o formulário estiver habilitado, ou seja, apenas antes de ser enviado.

```
$stars.on('mouseenter', function() {
  highlightStars($(this));
});

$stars.on('mouseleave', function() {
  highlightStars($('.review input:checked + label'));
});
```

Esses três blocos são responsáveis pelos eventos do mouse. O primeiro evento, o `mouseenter`, ocorre quando o usuário posiciona o mouse em cima de uma estrela. Nesse momento, vamos destacar a estrela clicada e as anteriores.

No evento `mouseleave`, vamos voltar ao estado inicial do formulário, ou seja, se já havia uma estrela marcada, tornamos ela marcada novamente.

Finalmente, temos o seguinte bloco:

```
$('.review input').on('change', function() {
  $stars.off('mouseenter').off('mouseleave').off('click');
  $(this).parent('form').submit();
});
```

O evento `change` é executado após o *click* em uma estrela. Nesse momento, desligamos todos os *Event handlers* que criamos, ou seja, desabilitamos a animação de estrelas e ativamos o evento de `submit` do formulário, fazendo o envio do formulário via AJAX.

12.9 Encerrando

Parabéns! Você perseverou até o fim das funcionalidades principais do Colcho.net! Foi um caminho longo e difícil, especialmente nesse capítulo. Mas você aprendeu a fazer **muita** coisa:

- Associações muitos-para-muitos;
- Remover objetos com segurança;
- Usar fontes e webfonts para deixar seu design bonito e escalonável;
- Customizar o Assets Pipeline;
- Novas funcionalidades do `ActiveRecord`: `find_or_initialize_by_...` e `find_or_create_by_...`;
- Organização de controles complexos;
- Organização de rotas complexas;
- Usar o `ActiveRecord` para fazer contas da melhor maneira possível;
- Opções avançadas de I18n;

- Extração e organização de templates complexos;
- Uso de *presenters* para simplificar templates complexos;
- Usar jQuery e AJAX;

Depois de tudo isso, você já está preparado para criar sua própria aplicação do zero. Os conhecimentos vistos nesse capítulo já cobrem muitas funcionalidades usadas em aplicações de verdade, com bastante complexidade. É lógico que livro nenhum irá substituir a experiência de construir as suas próprias aplicações, mas agora você já tem a base para colocar suas ideias em prática.

No próximo capítulo, vamos colocar algumas funcionalidades muito úteis, comuns na maioria das aplicações web e que serão de grande facilidade de implementar. Parabéns, você está chegando lá!

Capítulo 13

Polindo o Colcho.net

Não importa se você vai devagar, o que importa é você nunca parar
– Confúcio

Este é o último capítulo em que vamos desenvolver o Colcho.net! Você está chegando ao final, e a aplicação já está bem funcional. Nós vamos fazer algumas melhorias na aplicação em geral, desenvolvendo as seguintes funcionalidades: busca textual, *URL slugs* usando a gem `friendly_id`, paginação usando a *gem* `kaminari`, *upload* de fotos usando a *gem* `carrierwave` e, finalmente, colocar o Colcho.net online!

13.1 Faça buscas textuais apenas com o Rails

Até o momento, os usuários não conseguem pesquisar por nenhuma informação em nossa aplicação. Precisamos então permitir que o usuário pesquise informações no Colcho.net, melhorando a usabilidade da aplicação.

No Colcho.net há três informações importantes pelas quais um usuário pode pesquisar: localidade, título e a descrição de um quarto. Precisamos de uma busca que nos permita pesquisar nesses três "textos", ou seja, precisamos de uma busca textual.

Busca textual é um assunto **bastante** complexo e existem vários livros sobre o assunto, portanto vamos cobrir o que é possível criar com apenas o uso do Rails.

A funcionalidade é a seguinte: vamos criar um campo de texto próximo ao título de listagem de quartos. Vamos pegar o conteúdo da caixa de texto e procurar nos campos mencionados anteriormente. Dos resultados encontrados, vamos marcar visualmente para o usuário facilmente identificar o trecho que está procurando.

Primeiro, vamos construir a funcionalidade no *back-end*. Para isso, vamos criar um método chamado `.search` no modelo de quartos (`app/models/room.rb`):

```
class Room < ActiveRecord::Base
  # ...
  def self.search(query)
    if query.present?
      where(['location LIKE :query OR
              title LIKE :query OR
              description LIKE :query', :query => "%#{query}%"])
    else
      scoped
    end
  end
  # ..
end
```

Neste método, se a busca estiver presente, fazemos o filtro usando o operador `LIKE` do SQL para fazer a busca nos campos mencionados: `location`, `title` e `description`. Caso contrário, vamos retornar o escopo atual:

```
Room.search('Sao')
# SELECT "rooms".* FROM "rooms" WHERE (location LIKE '%Sao%' OR
#  title LIKE '%Sao%' OR
#  description LIKE '%Sao%')
# => [#<Room id: 4, ... >]

Room.search('')
# SELECT "rooms".* FROM "rooms"
# => [#<Room id: 4, ... >]
```

> **POR QUE USAR SCOPED E NÃO ALL?**
>
> Seria natural pensar em usar o método `.all` no caso da busca não estiver presente, pois retorna todos os objetos. Porém, se este método for usado em conjunto com outros escopos, o que é bastante comum, o `.all` iria cancelar todos os outros escopos e o comportamento final seria bem inesperado. O `.scoped` faz esse papel de manter o escopo.

Como o modelo já faz todo o trabalho pesado, no controle de quartos (`app/controllers/rooms_controller.rb`) vamos apenas chamar o método que acabamos de criar. A ação `index` fica da seguinte forma:

```
class RoomsController < ApplicationController
  # ...

  def index
    @search_query = params[:q]

    rooms = Room.search(@search_query)
    @rooms = rooms.most_recent.map do |room|
      RoomPresenter.new(room, self, false)
    end
  end

  # ...
end
```

Vamos diferenciar o *template* de índice (`app/views/rooms/index.html.erb`) apenas para mostrar um título diferente caso estejamos exibindo resultado de buscas. Para isso, vamos verificar a presença de um termo de busca (`search_query`). Vamos incluir no topo do *template* o formulário de busca. O resultado é:

```
<%= render 'search' %>

<h1>
  <% if @search_query.present? %>
    <%= t '.search_results' %>
  <% else %>
    <%= t '.title' %>
```

```
  <% end %>
</h1>

<%= render @rooms %>
```

Na *partial* contendo o formulário (`app/views/rooms/_search.html.erb`) não será possível usar o *helper* `form_for`, pois não estamos criando um formulário de modelo, mas sim um formulário qualquer. O Rails possui também *helpers* para esta situação:

```
<%= form_tag rooms_path, :method => :get, :class => 'search' do %>
  <%= text_field_tag :q, @search_query,
        :placeholder => t('.search_for') %>
<% end %>
```

O que fazemos é simplesmente enviar à ação `index` do controle de quartos o parâmetro `q`, que possui o conteúdo para filtrar a listagem. Observe que temos que declarar o método HTTP para `GET` para que o roteador não nos envie para a ação `create`.

Atualizamos as chaves I18n (`config/locales/pt-BR.yml`), para adicionar as mensagens relativas às buscas:

```
rooms:
  index:
    title: 'Quartos disponíveis'
    search_results: 'Resultados da busca'
  search:
    search_for: 'Buscar por...'
```

Em seguida, usamos o *helper* `highlight` do Rails, para destacar os resultados da busca, na *partial* de quarto. Alteraremos o título, a descrição e a localidade do quarto (`app/views/rooms/_room.html.erb`):

```
<article>
...
  <h2>
    <%= link_to highlight(room.title, @search_query), room.route %>
  </h2>
...
  <p>
    <span class="location">
```

```
      <%= link_to highlight(room.location, @search_query),
                  "https://maps.google.com/?q=#{room.location}",
                  :target => :blank %>
    </span>
  </p>
  <p><%= highlight(room.description, @search_query) %></p>
...
</article>
```

Por fim, adicionamos o estilo CSS para o campo de busca. Usamos a "Font Awesome" para adicionar o clássico ícone da lupa (app/assets/stylesheets/room.css.scss), e vamos estilizar os campos destacados pela função `highlight`:

```
#content .search {
  float: right;
  margin: 0;
  position: relative;
  &:after {
    padding: 5px;
    font-size: 18px;
    font-family: 'FontAwesome';
    position: absolute;
    content: "\f002";
    color: #bbb;
    top: 0;
    right: 0;
  }
}

.highlight {
  font-size: inherit;
  font-weight: inherit;
  background-color: gold;
}
```

Com essas alterações, ao realizarmos uma pesquisa, seu resultado é exibido com o destaque no termo procurado.

Figura 13.1: Resultado de buscas destacados

Por fim, é necessário lembrar que essa busca é uma busca simples. Se você quiser fazer buscas mais rebuscadas, é interessante usar tecnologias como o Solr (http://lucene.apache.org/solr/). Com o Solr, você é capaz de criar índices mais interessantes, fazer busca facetada, filtros e outras várias coisas. Se você tiver interesse, verifique a *gem* `sunspot` (https://github.com/outoftime/sunspot), que integra o Rails com o Solr.

13.2 URLs mais amigáveis através de slugs

Quando você acessa o site da Casa do Código, para ver esse livro, você entra no endereço http://casadocodigo.com.br/products/ruby-on-rails-coloque-sua-aplicacao-web-nos-trilhos. Internamente, esse livro possui um ID, e poderia ser usado nessa mesma URL, o ID ao invés do nome do livro, ficando: http://casadocodigo.com.br/products/3.

Note que a primeira URL possui um significado claro, enquanto a segunda, só olhando para a URL, não conseguimos dizer qual produto será acessado. Chamamos a primeira abordagem, com o nome do livro na URL, de *URL Slug*.

URL slugs são URLs que não usam IDs e sim uma URL mais interessante. Ou seja, ao invés de usar algo como http://localhost:3000/rooms/4, usamos http://localhost:3000/rooms/colchonete-na-sala. Esta técnica tem vantagens na otimização para mecanismos de busca (ou *SEO - Search Engine Optimization*).

Para implementar essa funcionalidade, podemos alterar a forma que as *URL helpers* do Rails constroem as rotas a partir do modelo. Os *helpers* chamam um método

chamado #to_param:

```
room = Room.first
# => #<Room id: 4, ... >

room.to_param
# => "4"
```

Uma alteração simples para transformar as URLs é sobrescrever o método #to_param para fazer o que quisermos e atualizar os controles de forma a usar a busca, ou seja, ao invés de usar .find no modelo, usamos algum outro método apropriado.

Fazer isso não é difícil. Porém, há vários detalhes que temos que tomar cuidado para implementar essa funcionalidade. Primeiro, temos que tomar cuidado com *slugs* antigas, ou seja, quando o usuário alterar o modelo de forma que o *slug* seja alterado, precisamos guardar o antigo *slug* para que links antigos não quebrem, sejam eles seus ou de outros sites. Segundo, o que fazer com conflitos de *slugs*? Mais ainda, como lidar com detalhes difíceis de prever, tal como transliteração de caracteres, ou seja, transformar símbolos como "ø" ou "ã" em "oe" e "a"?

Portanto, ao invés de ter que nos preocupar com estas situações (e outras não pensadas também), podemos nos aproveitar da experiência de outros desenvolvedores.

Para resolver o problema de *slugs*, vamos usar a *gem* friendly_id, feita pelo Norman Clarke, desenvolvedor que atua na comunidade Ruby e Rails há anos.

Para instalar esta *gem*, o primeiro passo é declará-la no Gemfile:

```
gem 'friendly_id'
```

E pedir para o Bundler instalar as dependências:

```
$ bundle
...
Installing friendly_id (4.0.8)
...
```

Em seguida, a *gem* precisa que você crie um campo extra, chamado slug, na tabela quarto, que é a informação que teremos a URL amigável. Para isso, vamos criar uma nova migração:

```
$ rails g migration add_slugs_to_rooms slug:string:index
 invoke  active_record
 create    db/migrate/20120811170119_add_slugs_to_rooms.rb
```

Precisamos adicionar um índice de unicidade. Por isso, alteraremos a migração gerada para adicionar essa restrição:

```
class AddSlugsToRooms < ActiveRecord::Migration
  def change
    add_column :rooms, :slug, :string
    add_index :rooms, :slug, :unique => true
  end
end
```

Vamos também ter que criar uma tabela para guardar os *slugs* antigos. O `friendly_id` já faz isso para nós, basta executar:

```
$ rails generate friendly_id
  create  db/migrate/20120811171412_create_friendly_id_slugs.rb
```

Executamos as migrações:

```
$ rake db:migrate
```

Por fim, alteraremos o modelo quarto (`app/models/room.rb`) para usar a funcionalidade de *slugs*, estendendo o módulo `FriendlyId`, adicionando uma restrição de presença de *slug* e configurando o módulo `friendly_id` para usar as funcionalidades `:slugged`, que é o modo padrão de operação da *gem* e a `:history`, para gravar o histórico de *slugs*:

```
class Room < ActiveRecord::Base
  extend FriendlyId

  #...
  validates_presence_of :title
  validates_presence_of :slug

  friendly_id :title, :use => [:slugged, :history]

  #...
end
```

Pronto! Você já tem a funcionalidade de *URL slugs*. Para testar, crie um novo quarto e veja a URL como fica (lembre-se de reiniciar o servidor do Rails, caso já não tenha feito).

> **Atualizando quartos já cadastrados**
>
> O `FriendlyId` irá funcionar com quartos sem *slug*, ou seja, irá usar o ID do modelo caso não seja possível usar o *slug*. Porém, para termos consistência, podemos atualizar os modelos para usarem *slugs*, basta ativar os *callbacks* de `save` que o `FriendlyId` irá fazer o resto:
>
> ```
> Room.find_each(&:save)
> ```
>
> O `find_each` é uma alternativa ao `all` para fazer alterações em lotes, caso você tenha muitos registros no banco de dados.

13.3 Paginação de dados de forma descomplicada

Imagine o Colcho.net no futuro, explodindo de sucesso e com mais de 500 quartos cadastrados. A listagem ficará lenta, pois são muitos objetos para se exibir e calcular a média da pontuação. Para essa situação, é normal paginar o resultado das buscas, de modo que os resultados menos interessantes fiquem ao final.

Essa funcionalidade não é complicada de se criar usando escopos. Mas a *gem* `kaminari` torna ainda mais fácil o nosso trabalho.

Vamos primeiro instalar a *gem*. Como de praxe, basta colocá-la no `Gemfile` e fazer o `bundle`:

```
gem 'kaminari'
```

```
$ bundle
...
Installing kaminari (0.13.0)
...
```

O `kaminari` insere, através de meta-programação, alguns métodos no `ActiveRecord` para fazer a paginação, porém o nosso uso de *presenters* atrapalhou isso, já que estamos usando uma Array deles.

Precisamos criar um outro *presenter* para que possamos chamar os métodos de paginação do `kaminari`. Outra coisa que precisamos fazer é tornar essa nova classe uma espécie de coleção, para que o `render :collection => @rooms` continue funcionando.

Infelizmente o Rails não definiu uma interface para este tipo de interação. Portanto precisamos implementar o método `#to_ary`, que faz uma conversão do objeto atual para Array quando isso for necessário (via conversão implícita). Veja o exemplo a seguir:

```
class ImplicitArray
  def to_ary
    [1,2,3]
  end
end
# => nil
[:a] + ImplicitArray.new
# => [:a, 1, 2, 3]
```

Repare que ao juntarmos o Array de `:a`, com a instância de `ImplicitArray`, o resultado foi um novo Array com a junção de `:a` e o resultado do método `to_ary` de `ImplicitArray`.

Com isso em mente, vamos criar o *presenter* `RoomCollectionPresenter` (`app/presenters/room_collection_presenter.rb`), que irá delegar os métodos de paginação para a coleção `ActiveRecord` e criar *presenters* de quarto quando o *collection presenter* for convertido para Array.

```
class RoomCollectionPresenter
  delegate :current_page, :num_pages, :limit_value,
    :to => :@rooms

  def initialize(rooms, context)
    @rooms = rooms
    @context = context
  end

  def to_ary
    @rooms.map do |room|
      RoomPresenter.new(room, @context, false)
    end
  end
end
```

Reuso dos presenters

Este é o tipo de código que sempre precisamos, portanto você pode aplicar em seus próprios projetos. Se preferir usar um *presenter* funcional e pronto, vale a pena checar a *gem* `simple_presenter`, do Nando Vieira: https://github.com/fnando/simple_presenter

Alteramos a ação `index` do `RoomsController` (`app/controllers/rooms_controller.rb`) para que ela fique da seguinte maneira:

```
class RoomsController < ApplicationController
  PER_PAGE = 10

  # ...

  def index
    @search_query = params[:q]
    rooms = Room.search(@search_query).
      page(params[:page]).
      per(PER_PAGE)

    @rooms = RoomCollectionPresenter.new(rooms.most_recent, self)
  end
end
```

Aplicamos a paginação no modelo de quarto, via o uso dos métodos `.page`, que define a página a ser buscada e o método `.per`, que define a quantidade de objetos por página. Na última linha da ação, embrulhamos a coleção no *presenter* que acabamos de criar.

Com isso pronto, para usar a paginação do `kaminari`, basta colocar o *helper* `paginate` no *template* da ação `index` (`app/views/rooms/index.html.erb`):

```
...
<%= render @rooms %>

<%= paginate @rooms %>
```

Isso é tudo que precisamos para fazer a paginação funcionar.

Vamos melhorar um pouco mais, criando um novo conjunto de chaves I18n para o `kaminari`, no arquivo `config/locales/pt-BR.pagination.yml`:

```
pt-BR:
  views:
    pagination:
      first: "&laquo; Primeira"
      last: "Última &raquo;"
      previous: "&lsaquo; Anterior"
      next: "Próxima &rsaquo;"
      truncate: "..."
```

Por fim, um pouco de CSS para ajustar a exibição da paginação (`app/assets/stylesheets/default.css.scss`):

```
.pagination {
  margin: 20px 0;
}
.pagination span {
  padding: 5px;
  margin: 0 3px;
  background-color: #fff;
  border: 1px solid #ccc;
  @include shadow(#ccc, 0, 3px, 6px);
}
```

O resultado é o seguinte:

Figura 13.2: Links para paginação no fim da página

Pronto, agora você tem uma paginação feita, sem trazer dados demais para a memória e sem exibir informações desnecessárias na tela.

13.4 Upload de fotos de forma simples

Imagine você alugando um quarto ou um apartamento sem conseguir ver fotos desse lugar? Difícil, não? Para isso, precisamos permitir que fotos sejam associadas ao quarto, através de upload de arquivos de imagens.

Para fazer o *upload* de fotos de quartos, vamos usar uma *gem* chamada `carrierwave`. Com ela é possível criar *thumbnails* de fotos automaticamente e até enviar essa foto para ser servida de serviços como o S3, da Amazon (http://aws.amazon.com/s3/) ou Cloud Files, da Rackspace (http://www.rackspace.com/cloud/public/files/), serviços bastante interessantes para grandes sites.

Para instalá-la, a receita é parecida: colocar a *gem* no `Gemfile` e fazer o `bundle`. A diferença é que temos que declarar a dependência à gem `rmagick` manualmente. Isso deve-se ao fato de que o `carrierwave` funciona com outras *gems* também:

```
gem 'carrierwave'
gem 'rmagick'
```

```
$ bundle
...
Installing carrierwave (0.6.2)
Installing rmagick (2.13.1) with native extensions
...
```

O funcionamento do `carrierwave` é o seguinte: primeiro, é necessário criar um *uploader*, uma classe com a descrição das transformações que a imagem vai passar (criação de *thumbnails*, por exemplo), onde guardar o arquivo enviado pelo usuário e outras coisas. Para facilitar esse trabalho, o `carrierwave` instala um gerador. Então, vamos criar o *uploader* para fotos de quartos:

```
$ rails g uploader Picture
 create  app/uploaders/picture_uploader.rb
```

O arquivo gerado possui diversos comentários sobre a forma de utilizar o *uploader* e os parâmetros configuráveis. O resultado, das linhas não comentadas deve ser o seguinte:

```
# encoding: utf-8

class PictureUploader < CarrierWave::Uploader::Base
  include CarrierWave::RMagick
```

```
  include Sprockets::Helpers::RailsHelper

  storage :file

  # Diretório onde os arquivos serão armazenados
  def store_dir
    "uploads/#{model.class.to_s.underscore}/#{mounted_as}/#{model.id}"
  end

  # Redimensiona a imagem para ficar no tamanho de
  # no máximo 500x500, mantendo o aspecto e cortando
  # a imagem, se necessário.
  process :resize_to_fill => [500, 500]

  # Dimensões do thumbnail
  version :thumb do
    process :resize_to_fill => [100, 100]
  end

  # Informa os formatos permitidos
  def extension_white_list
    %w(jpg jpeg gif png)
  end
end
```

Carrierwave não é só para imagens

Apesar de conter muitas facilidades para o *upload* de imagens, ele pode ser usado com qualquer outro formato de arquivo, desde que você desative as funcionalidades específicas para imagens, como processamento.

Uma vez criado o *uploader*, precisamos criar uma coluna no banco de dados para que o `carrierwave` guarde o nome do arquivo e saiba recuperá-lo na hora de exibir a foto. Para isso, criemos e executemos a migração a seguir:

```
$ rails g migration add_picture_to_rooms picture
  create    db/migrate/20120813014045_add_picture_to_rooms.rb
```

```
$ rake db:migrate
==  AddPictureToRooms: migrating ==============
-- add_column(:rooms, :picture, :string)
   -> 0.0012s
==  AddPictureToRooms: migrated (0.0013s) =====
```

Em seguida, precisamos associar o *uploader* ao modelo quarto. Isso é feito através da *class macro* `mount`, método que o `carrierwave` adiciona ao `ActiveRecord`. Portanto, no modelo quarto (`app/models/room.rb`), basta adicionar o seguinte código:

```
class Room < ActiveRecord::Base
  # Adicione :picture na lista de atributos:
  attr_accessible :description, :location, :title, :picture

  # ...

  mount_uploader :picture, PictureUploader
  friendly_id :title, :use => [:slugged, :history]

  # ...
end
```

Essa *class macro* irá tornar o campo `picture` do modelo quarto em um `PictureUploader`, ao invés de uma simples `string`. A partir daí, basta colocarmos mais um campo no formulário de quartos, para que o arquivo seja informado. Então, no `app/views/rooms/_form.html.erb`:

```
...
<%= form_for(@room) do |f| %>
  ...
  <p>
    <%= f.label :picture %>
    <%= f.file_field :picture %>
    <%= error_tag @room, :picture %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>
```

Adicione também a chave para traduzir o novo campo no arquivo de I18n (`config/locales/pt-BR.yml`):

```
pt-BR:
  #...
  activerecord:
    #...
    attributes:
      # ...
      room:
        description: Descrição
        location: Localização
        title: Título
        picture: Foto
```

Por fim, vamos alterar o *presenter*, HTML e o CSS para exibir a imagem. Altere o *presenter* de quartos para incluir os métodos que verificam se há uma imagem, e também que devolve o *thumbnail* e a própria imagem (`app/presenters/room_presenter.rb`):

```
class RoomPresenter
  # ...
  def picture_url
    @room.picture_url
  end

  def thumb_url
    @room.picture.thumb.url
  end

  def has_picture?
    @room.picture?
  end
end
```

O *template* de quartos (`app/views/rooms/_room.html.erb`) fica assim:

```
<article class="room">
  ...

  <%= link_to(image_tag(room.thumb_url), room.picture_url)
                                       if room.has_picture? %>
```

```
  <p><%= highlight(room.description, @search_query) %></p>
  ...
</article>
```

O CSS para a imagem de quartos (`app/assets/stylesheets/room.css.scss`) é:

```
.room img {
  float: left;
  margin-right: 10px;
}
```

Et voilà! Os quartos agora possuem foto:

Figura 13.3: Quarto com uma foto

13.5 Coloque a aplicação no ar com o Heroku

Antigamente, colocar um aplicativo web no ar era complicado e exigia bastante conhecimento de configuração de servidor. Hoje em dia, contudo, existem serviços que oferecem hospedagem de aplicativos sem a noção de termos um servidor. *Platform as a service - PaaS*, ou "Plataforma como serviço", é mais uma das facetas de *Cloud Computing*, na qual um usuário contrata uma plataforma e poder de processamento, ao invés de contratar servidores, sejam de metal ou virtual.

Um desses serviços é o Heroku (www.heroku.com). Hoje em dia, não existe forma mais fácil de colocar uma aplicação Rails no ar do que usar o Heroku. Por isso, vamos usá-lo: é de graça para pouco processamento de requisição (1 requisição por segundo), que é o bastante para nossa aplicação.

Preparando a aplicação para o Heroku

Porém, precisamos fazer algumas alterações no aplicativo para fazer o Heroku funcionar: o Heroku funciona com PostgreSQL e estamos usando o SQLite3. Por esse motivo, é necessário colocar a *gem* `pg` no `Gemfile`, responsável pela conectividade a bancos PostgreSQL e alterar uma consulta SQL. Primeiro, vamos a alteração do `Gemfile` e execute `bundle` em seguida:

```
# gem 'sqlite3'
gem 'pg'
```

```
$ bundle
...
Installing pg (0.14.0) with native extensions
...
```

Vamos agora alterar a consulta SQL problemática: no PostgreSQL, a consulta `LIKE` é sensível a maiúsculas e minúsculas, enquanto a `ILIKE` não. Vamos alterar o método `Room.search` (`app/models/room.rb`) para usar a nova consulta:

```
class Room < ActiveRecord::Base
  # ...

  def self.search(query)
    if query.present?
      where(['location ILIKE :query OR
              title ILIKE :query OR
              description ILIKE :query', :query => "%#{query}%"])
    else
      scoped
    end
  end

  # ...
end
```

Desenvolvimento usando PostgreSQL

Infelizmente esse comportamento do `LIKE` vs. `ILIKE` é incompatível no SQLite. Por isso, pode ser interessante para você desenvolver diretamente no PostgreSQL.

Se você usa Linux, verifique os pacotes de sua distribuição. De preferência, instale uma versão igual ou superior à 9. No OS X, para usuários de 10.7 (Lion) ou superior, você já tem o PostgreSQL. Para Windows, você pode usar o instalador do site oficial: http://postgresql.org.

Em seguida, altere a parte `development` e `test` do arquivo `config/database.yml` para refletir as alterações. Não é necessário ter `production`, ele será criado pelo próprio Heroku. Veja o exemplo de como deve ficar o arquivo:

```
development:
  adapter: postgresql
  database: colchonet_dev
  host: localhost
  username: vinibaggio
  password:
  pool: 5
  timeout: 5000


test:
  adapter: postgresql
  database: colchonet_test
  host: localhost
  username: vinibaggio
  password:
  pool: 5
  timeout: 5000
```

Por fim, execute:

```
$ rake db:create db:migrate
==  CreateRooms: migrating =====================
-- create_table(:rooms)
...
==  AddPictureToRooms: migrating ===============
-- add_column(:rooms, :picture, :string)
   -> 0.0011s
==  AddPictureToRooms: migrated (0.0012s) ======
```

Colocando a aplicação no ar

Para começar a usar o Heroku, é necessário instalar o `git`, criar uma conta no Heroku e baixar o Heroku Tools. Uma vez com tudo instalado, basta ir na pasta do projeto, e criar um repositório `git` e fazer um *commit*:

```
$ git init .
Initialized empty Git repository in /book/code/colchonet/.git/

$ git add .

$ git commit -m "Primeiro commit"
[master (root-commit) a2ed4be] Primeiro commit
 106 files changed, 3169 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 Gemfile
 create mode 100644 Gemfile.lock
 create mode 100644 README.rdoc
 ...
```

> **O que é git? Como instalar?**
>
> O `git` é um sistema de versionamento de código-fonte. Se você já ouviu falar em CVS, SVN ou Mercurial, o `git` é parecido com todos eles, com algumas outras funcionalidades bastante interessantes. Para saber mais, veja o *screencast* "Começando com Git" http://colcho.net/comecando-com-git. Nele você poderá saber como funciona o `git` e como instalá-lo.

Uma vez com o *commit* criado, vamos usar o `heroku tools` para criar um repositório remoto chamado `heroku` e preparar o nosso novo site.

```
$ heroku login
Enter your Heroku credentials.
Email: XXX@XXX.com
Password (typing will be hidden):
Authentication successful.

$ heroku create
```

```
Creating calm-badlands-8648... done, stack is cedar
http://calm-badlands-8648.herokuapp.com/ |
             git@heroku.com:calm-badlands-8648.git
Git remote heroku added
```

Precisamos adicionar um *add-on* ao novo site para que possamos entregar os e-mails de cadastro. Vamos usar o Mailgun, é de graça e possui integração simples com o Heroku:

```
$ heroku addons:add mailgun:starter
Adding mailgun:starter on calm-badlands-8648... done, v8 (free)
Use `heroku addons:docs mailgun:starter` to view documentation.
```

Precisamos atualizar o ambiente `production` (`config/environments/production.rb`), pois é neste ambiente que o Heroku executa a nossa aplicação. Precisamos atualizar as configurações do `ActionMailer` para usar a nova URL do site e as configurações do Mailgun (conforme vimos na seção 9.2):

```
Colchonet::Application.configure do
  #...

  config.action_mailer.default_url_options = {
    :host => "calm-badlands-8648.herokuapp.com"
  }

  config.action_mailer.smtp_settings = {
    :port           => ENV['MAILGUN_SMTP_PORT'],
    :address        => ENV['MAILGUN_SMTP_SERVER'],
    :user_name      => ENV['MAILGUN_SMTP_LOGIN'],
    :password       => ENV['MAILGUN_SMTP_PASSWORD'],
    :domain         => 'calm-badlands-8648.herokuapp.com',
    :authentication => :plain,
  }

  config.action_mailer.delivery_method = :smtp
end
```

As configurações do Mailgun são colocadas como variáveis de ambiente, bastante conveniente para não termos que gerenciar chaves de API. Em seguida, execute o derradeiro comando:

```
$ git push heroku master
Counting objects: 154, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (140/140), done.
Writing objects: 100% (154/154), 940.50 KiB | 339 KiB/s, done.
Total 154 (delta 10), reused 0 (delta 0)

-----> Heroku receiving push
...
-----> Discovering process types
       Procfile declares types      -> (none)
       Default types for Ruby/Rails -> console, rake, web, worker
-----> Compiled slug size is 12.4MB
-----> Launching... done, v4
       http://calm-badlands-8648.herokuapp.com deployed to Heroku

To git@heroku.com:calm-badlands-8648.git
 * [new branch]      master -> master
```

Quase pronto... Por fim, temos que migrar o banco de dados recém criado:

```
$ heroku run rake db:migrate
Running `rake db:migrate` attached to terminal... up, run.1
Connecting to database specified by DATABASE_URL
Migrating to CreateRooms (20120610045608)
...
```

Depois que o comando terminar, basta acessar a URL que você recebeu e já pode passar para sua família e amigos! Muito bom, não é mesmo? Parabéns! Você completou o Colcho.net!

Capítulo 14

Próximos passos

Agora, isso não é o fim. Nem sequer é o início do fim. Mas é, talvez, o fim do começo.
– Sir Winston Churchill

Agora que você já é familiar ao Rails, está na hora de fazer as suas próprias aplicações. Seria muita pretensão deste humilde livro te ensinar tudo que existe no ecossistema Rails, portanto, depois da leitura deste livro, ainda é necessário buscar mais material.

Lista de email

Se você quer tirar alguma dúvida sobre este livro, você pode se juntar à lista de emails do livro em http://colcho.net/lista e poderá enviar sua pergunta. Os participantes dela, inclusive o autor deste livro, tentarão te ajudar. Lembre-se de ser educado!

Conhecimentos de Rails

O primeiro lugar que você deve buscar para tirar dúvidas e entender o funcionamento específico de algum componente do Rails são os RailsGuides (http://guides.rubyonrails.org). Nele você poderá buscar informações sobre os principais componentes do Rails, tais como `ActiveRecord`, *templates*, etc.

Se sua dúvida for mais específica, talvez seja interessante consultar a documentação da API, que fica em http://api.rubyonrails.org. A documentação é fácil de navegar, bastando que você saiba o método que quer procurar.

Se você quiser aprender receitas de como resolver certos problemas, o Ryan Bates, bastante famoso na comunidade Rails, faz o RailsCasts (http://railscasts.com). Cada *screencast* é uma receita de como resolver um problema, e o inglês que ele fala é claro, sendo uma ótima maneira de aprender. Se você ainda não está confortável em entender uma narração em inglês, pode ler os AsciiCasts (http://asciicasts.com/), que é praticamente uma transcrição do RailsCasts.

Se você prefere livros, uma grande recomendação é o "Rails 3 Recipes", do Chad Fowler (http://colcho.net/rails-recipes). Nele você pode encontrar diversas receitas de bolo de como resolver problemas encontrados no dia-a-dia. Se você quer se aprofundar no *framework*, não existe livro melhor do que o "Crafting Rails Applications" (http://colcho.net/crafting-rails-apps). Neste livro, o José Valim, um dos principais desenvolvedores do próprio *framework*, te guia para um mergulho de cabeça nas profundidades do Rails. Pode ser um pouco difícil para iniciantes, mas é um livro de altíssima qualidade.

Dominando o Ruby

Se você quer se tornar um desenvolvedor Ruby e Rails profissional, recomendo aprofundar-se na linguagem, pois é de extrema importante um profissional conhecer bem as ferramentas que está trabalhando. Minha recomendação para este fim é o "Eloquent Ruby", do Russ Olsen (http://colcho.net/eloquent-ruby). Outro livro recomendado é o "Ruby Programming Language" (http://colcho.net/ruby-programming-lang), de autoria de David Flanagan e Yukihiro Matsumoto, o próprio Matz.

Dominando testes

A comunidade Ruby e Rails em geral gosta bastante de testes unitários e argumentam que esta prática melhora o *design* de código e ajuda a reduzir o número de

bugs. É interessante entender as ideias, por isso recomendo a leitura do "Test-Driven Development", do Kent Beck (http://colcho.net/tdd). Em seguida, é interessante ler como TDD se aplica no Ruby, com RSpec e Cucumber, através do livro "The RSpec Book" (http://colcho.net/rspec-book). Você pode também procurar o "Guia rápido de RSpec", do Nando Vieira (http://colcho.net/guia-rspec).

É difícil no começo, não se preocupe se tiver dificuldades, é normal. Um pouco de perseverança e você poderá sentir os benefícios dessa prática. Há também um livro específico para receitas de teste com Rails, o "Rails Test Prescriptions" (http://colcho.net/test-prescriptions).

Envolvimento na comunidade

A comunidade rubista brasileira é bastante ativa. Em São Paulo, por exemplo, existe o GURU-SP, porém é possível encontrar outros grupos de Ruby e Rails pelo Brasil e pelo mundo. Procure as listas de email, participe e fique atento aos encontros, você pode aprender muito! Não deixe de participar também em outros eventos de programação que existem no Brasil, pois exposição a tecnologias e novas ideias sempre te ajudarão com qualquer linguagem ou *framework*.

Open source e leitura de código

A comunidade de Ruby e Rails é bastante envolvida em *open source* e se envolver em um projeto é uma grande forma de, além de contribuir, aprender com outros desenvolvedores. A leitura de código também é encorajada. Uma ótima maneira é procurar sua *gem* favorita no GitHub (http://www.github.com) e navegar pelo código fonte.

Índice Remissivo

Referências Bibliográficas

[1] David Thomas Andrew Hunt. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, 1999.

[2] Kent Beck. *Test-Driven Development: By Example*. Addison-Wesley Professional, 2003.

[3] Gregory Brown. *Ruby Best Practices*. O'Reilly, 2009.

[4] Gregory Brown. Issue 24: Connascence as a software design metric. http://colcho.net/issue-24, 2011.

[5] Zach Dennis Aslak Hellesøy Bryan Helmkamp Dan North David Chelimsky, Dave Astels. *The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends*. Pragmatic Bookshelf, 2010.

[6] Avdi Grimm. *Exceptional Ruby*. ShipRise, 2011.

[7] Ian Robinson Jim Webber, Savas Parastatidis. *Rest in Practice: Hypermedia and Systems Architecture*. O'Reilly Media, 2010.

[8] Paolo Perrotta. *Metaprogramming Ruby: Program like the Ruby Pros*. Pragmatic Bookshelf, 2010.

[9] Nando Vieira. Guia rápido de rspec. http://colcho.net/guia-rspec, 2010.

[10] Jim Weirich. Building blocks of modularity. http://colcho.net/bbom, 2009.